*Red Hat Enterprise Linux 4*

# Red Hat SELinux Guide

**Red Hat Enterprise Linux 4: Red Hat SELinux Guide**

Red Hat, Inc.

1801 Varsity Drive
Raleigh NC 27606-2072 USA
Phone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701
PO Box 13588
Research Triangle Park NC 27709 USA

rhel-selg(EN)-4-Print-RHI (2005-02-15-T16:20)

The GPG fingerprint of the security@redhat.com key is:
CA 20 86 86 2B D6 9D FC 65 F6 EC C4 21 91 80 CD DB 42 A6 0E

# Table of Contents

# Introduction to the Red Hat SELinux Guide

Welcome to the Red Hat SELinux Guide. This guide addresses the complex world of SELinux policy, and has the goal of teaching you how to understand, use, administer, and troubleshoot SELinux in a Red Hat Enterprise Linux environment. SELinux, an implementation of *mandatory access control* (MAC) in the Linux kernel, adds the ability to administratively define policies on all *subjects* (processes) and *objects* (devices, files, and signaled processes). These terms are used as an abstract when discussing actors/doers and their targets on a system. This guide commonly refers to processes, the source of an operations, and objects, the target of an operation.

This guide opens with a short explanation of SELinux, some assumptions about the reader, and an explanation of document conventions. The first part of the guide provides an overview of the technical architecture and how policy works, specifically the policy that comes with Red Hat Enterprise Linux called the *targeted* policy. The second part focuses on working with SELinux, including maintaining and manipulating your systems, policy analysis, and compiling your custom policy. Working with some of the daemons that are confined by the targeted policy is discussed throughout. These daemons are collectively called the *targeted daemons*.

One powerful way of finding information in this guide is the *Index*. The *Index* has direct links to sections on specific terminology, and also features lists of various SELinux syntaxes, as well as *what are/what is* and *how to* entries.

## 1. What Is SELinux?

This section is a very brief overview of SELinux. More detail is given in Part I *Understanding SELinux* and Appendix A *Brief Background and History of SELinux*.

*Security-enhanced Linux* (*SELinux*) is an implementation of a *mandatory access control* mechanism. This mechanism is in the Linux kernel, checking for allowed operations after standard Linux *discretionary access controls* are checked.

To understand the benefit of mandatory access control (MAC) over traditional discretionary access control (DAC), you need to first understand the limitations of DAC.

Under DAC, ownership of a file object provides potentially crippling or risky control over the object. A user can expose a file or directory to a security or confidentiality breach with a misconfigured `chmod` command and an unexpected propagation of access rights. A process started by that user, such as a CGI script, can do anything it wants to the files owned by the user. A compromised Apache HTTP server can perform any operation on files in the Web group. Malicious or broken software can have root-level access to the entire system, either by running as a root process or using `setuid` or `setgid`.

Under DAC, there are really only two major categories of users, administrators and non-administrators. In order for services and programs to run with any level of elevated privilege, the choices are few and course grained, and typically resolve to just giving full administrator access. Solutions such as ACLs (*access control lists*) can provide some additional security for allowing non-administrators expanded privileges, but for the most part a root account has complete discretion over the file system.

A MAC or *non-discretionary access control* framework allows you to define permissions for how all processes (called *subjects*) interact with other parts of the system such as files, devices, sockets, ports, and other processes (called *objects* in SELinux). This is done through an administratively-defined security policy over all processes and objects. These processes and objects are controlled through the kernel, and security decisions are made on all available information rather than just user identity. With this model, a process can be granted just the permissions it needs to be functional. This follows the principle of *least privilege*. Under MAC, for example, users who have exposed their data using `chmod` are protected by the fact that their data is a kind only associated with user home directories, and confined processes cannot touch those files without permission and purpose written into the policy.

SELinux is implemented in the Linux kernel using the LSM (*Linux Security Modules*) framework. This is only the latest implementation of an ongoing project, as detailed in Appendix A *Brief Background and History of SELinux*. To support fine-grained access control, SELinux implements two technologies: *Type Enforcement*™ (TE) and a kind of *role-based access control* (RBAC), which are discussed in Chapter 1 *SELinux Architectural Overview*.

Type Enforcement involves defining a *type* for every subject, that is, process, and object on the system. These types are defined by the SELinux *policy* and are contained in security labels on the files themselves, stored in the *extended attributes* (*xattrs*) of the file. When a type is associated with a processes, the type is called a *domain*, as in, "`httpd` is in the domain of `httpd_t`." This is a terminology difference leftover from other models when domains and types were handled separately.

All interactions between subjects and objects are disallowed by default on an SELinux system. The policy specifically allows certain operations. To know what to allow, TE uses a matrix of domains and object types derived from the policy. The matrix is derived from the policy rules. For example, `allow httpd_t net_conf_t:file { read getattr lock ioctl };` gives the domain associated with `httpd` the permissions to read data out of specific network configuration files such as `/etc/resolv.conf`. The matrix clearly defines all the interactions of processes and the targets of their operations.

Because of this design, SELinux can implement very granular access controls. For Red Hat Enterprise Linux 4 the policy has been designed to restrict only a specific list of daemons. All other processes run in an unconfined state. This policy is designed to help integrate SELinux into your development and production environment. It is possible to have a much more strict policy, which comes with an increase in maintenance complexity.

## 2. Prerequisites for This Guide

The technical skills required for this guide are not very extensive. The most important skill to have is an ability to learn technical theories and put them into practice. It helps if you come into this guide with an idea of what you want to do, such as administrating a set of common services, making user content from `/home/` served via Apache HTTP, manipulating policy to get a custom PHP Web application running, or writing a policy from to enable a custom application to be protected by SELinux. The following is helpful to have as you read through this guide:

- Strong working understanding of Linux, especially Red Hat Enterprise Linux.
- If you are going to be administrating services, manipulating or analyzing policy, junior- to mid-level system administration skills and experience is necessary, such as being a Red Hat Certified Technician (RHCT) or Red Hat Certified Engineer (RHCE)..

  To work with SELinux at that level, you must have the following:

  - An understanding of traditional Linux/UNIX security.
  - An understanding of how a Linux/UNIX system operates on a lower-level, such as how the kernel has system calls for various operations (open, close, read, write, ioctl, poll, etc.) An understanding of programming and system theory is useful in writing policy.
  - A familiarity with the m4 macro language, which is helpful in understanding some parts of the SELinux policy.
  - Read many of the NSA papers, listed in Chapter 9 *References*.
  - Administrator privileges on the system you have Red Hat Enterprise Linux installed on is necessary to perform many of the operations in this guide. However, there is plenty of useful information for end-users.

- Somewhere you can examine and work with the policy sources. This can be a test or development machine, or possibly a workstation. Many of the examples and explanations in this book assume that you have the system in front of you to explore while you read.
- Some additional patience. SELinux is a different way of handling access control than many administrators and users are familiar with.

Information about Red Hat training can be obtained via http://www.redhat.com/training/.

## 3. Conventions for SELinux Directories and Files

There are two main directories for SELinux policy in `/etc/selinux/`:

- `/etc/selinux/<policyname>/policy/` — the binary policy and runtime configuration files.
- `/etc/selinux/<policyname>/src/policy/` — policy sources.

It is possible to have more than one policy existing on the system, although only one may be loaded at a time. The policy binary files, and possibly source files, are located in `/etc/selinux/<policyname>/`, where *<policyname>* is the name of your policy, such as targeted, strict, webhost, test, and so forth. The configuration file `/etc/selinux/config` defines which policy is used, for example *SELINUXTYPE=targeted*.

In this document, the convention of `$DIRECTORY_TYPE` is used instead of the full path to assist in readability:

- The variable directory `$SELINUX_SRC/` is a substitute for the generic directory of `/etc/selinux/<policyname>/src/policy/` and the targeted policy source directory at `/etc/selinux/targeted/src/policy/`.
- The variable directory `$SELINUX_POLICY/` is a substitute for the generic directory of `/etc/selinux/<policyname>/policy/` and the binary targeted policy directory at `/etc/selinux/targeted/policy/`.

An important file is the audit log file. In Red Hat Enterprise Linux, `$AUDIT_LOG` by default is `/var/log/messages`. However, this is configurable via `/etc/syslog.conf`, and future work on an audit daemon will handle kernel audit events and log them into a separate file. Because of the variable nature of where the audit logs are, the variable file `$AUDIT_LOG` is used as a substitute.

Other important files and directories include `$SELINUX_POLICY/booleans` and `$SELINUX_POLICY/contexts/`, which are both discussed in Section 3.2 *Files and Directories of the Targeted Policy*.

The most important file for SELinux is the binary policy file. This file is located at `/etc/selinux/targeted/policy/policy.<XY>`. The *<XY>* represents the two digits of the policy version. In the case of Red Hat Enterprise Linux 4, this file is `policy.18`.

## 4. Document Conventions

When you read this manual, certain words are represented in different fonts, typefaces, sizes, and weights. This highlighting is systematic; different words are represented in the same style to indicate their inclusion in a specific category. The types of words that are represented this way include the following:

`command`

Linux commands (and other operating system commands, when used) are represented this way. This style should indicate to you that you can type the word or phrase on the command line and press [Enter] to invoke a command. Sometimes a command contains words that would be displayed in a different style on their own (such as file names). In these cases, they are considered to be part of the command, so the entire phrase is displayed as a command. For example:

Use the `cat testfile` command to view the contents of a file, named `testfile`, in the current working directory.

`file name`

File names, directory names, paths, and RPM package names are represented this way. This style should indicate that a particular file or directory exists by that name on your system. Examples:

The `.bashrc` file in your home directory contains bash shell definitions and aliases for your own use.

The `/etc/fstab` file contains information about different system devices and file systems.

Install the `webalizer` RPM if you want to use a Web server log file analysis program.

**application**

This style indicates that the program is an end-user application (as opposed to system software). For example:

Use **Mozilla** to browse the Web.

[key]

A key on the keyboard is shown in this style. For example:

To use [Tab] completion, type in a character and then press the [Tab] key. Your terminal displays the list of files in the directory that start with that letter.

[key]-[combination]

A combination of keystrokes is represented in this way. For example:

The [Ctrl]-[Alt]-[Backspace] key combination exits your graphical session and returns you to the graphical login screen or the console.

**text found on a GUI interface**

A title, word, or phrase found on a GUI interface screen or window is shown in this style. Text shown in this style is being used to identify a particular GUI screen or an element on a GUI screen (such as text associated with a checkbox or field). Example:

Select the **Require Password** checkbox if you would like your screensaver to require a password before stopping.

**top level of a menu on a GUI screen or window**

A word in this style indicates that the word is the top level of a pulldown menu. If you click on the word on the GUI screen, the rest of the menu should appear. For example:

Under **File** on a GNOME terminal, the **New Tab** option allows you to open multiple shell prompts in the same window.

If you need to type in a sequence of commands from a GUI menu, they are shown like the following example:

Go to **Applications** (the main menu on the panel) => **Programming => Emacs Text Editor** to start the **Emacs** text editor.

**button on a GUI screen or window**

This style indicates that the text can be found on a clickable button on a GUI screen. For example:

Click on the **Back** button to return to the webpage you last viewed.

`computer output`

Text in this style indicates text displayed to a shell prompt such as error messages and responses to commands. For example:

The `ls` command displays the contents of a directory. For example:

```
Desktop                about.html       logs          paulwesterberg.png
Mail                   backupfiles      mail          reports
```

The output returned in response to the command (in this case, the contents of the directory) is shown in this style.

`prompt`

A prompt, which is a computer's way of signifying that it is ready for you to input something, is shown in this style. Examples:

`$`

`#`

`[stephen@maturin stephen]$`

`leopard login:`

**`user input`**

Text that the user has to type, either on the command line, or into a text box on a GUI screen, is displayed in this style. In the following example, **`text`** is displayed in this style:

To boot your system into the text based installation program, you must type in the **`text`** command at the `boot:` prompt.

*<`replaceable`>*

Text used for examples, which is meant to be replaced with data provided by the user, is displayed in this style. In the following example, *<`version-number`>* is displayed in this style:

The directory for the kernel source is `/usr/src/kernels/`*<`version-number`>*`/`, where *<`version-number`>* is the version and type of kernel installed on this system.

Additionally, we use several different strategies to draw your attention to certain pieces of information. In order of how critical the information is to your system, these items are marked as a note, tip, important, caution, or warning. For example:

**Note**

Remember that Linux is case sensitive. In other words, a rose is not a ROSE is not a rOsE.

**Tip**

The directory `/usr/share/doc/` contains additional documentation for packages installed on your system.

**Important**

If you modify the DHCP configuration file, the changes do not take effect until you restart the DHCP daemon.

**Caution**

Do not perform routine tasks as root — use a regular user account unless you need to use the root account for system administration tasks.

**Warning**

Be careful to remove only the necessary partitions. Removing other partitions could result in data loss or a corrupted system environment.

## 5. Code Presentation Conventions

In addition to the standard document conventions covered in Section 4 *Document Conventions*, there are some additional conventions related specifically to discussing source code:

`classname`

This is the name of a class in an object-oriented (*OO*) programming language. For example, the class `com.arsdigita.categorization.CategoryTreeNode`.

`method name`

This is the name of a method in an OO programming language, e.g. the method `getBaseDataObjectType`.

`function`

The name of a function or subroutine, as in a programming language. For example, the function `SecurityLogger.warn()`.

`variable name`

The name of a variable. For example, the variable `BASE_DATA_OBJECT_TYPE`.

`option`

An option for a software command or `Method`. For example, a user has been granted `read` privileges on an object.

return value

The value returned by a function. For example, a method returns null.

program listing

A literal listing of all or part of a program:

```
extern void sem_exit (void);
extern struct task_struct *child_reaper;

int getrusage(struct task_struct *, int, struct rusage *);

static void __unhash_process(struct task_struct *p)
{
        nr_threads--;
        detach_pid(p, PIDTYPE_PID);
        detach_pid(p, PIDTYPE_TGID);
        if (thread_group_leader(p)) {
                detach_pid(p, PIDTYPE_PGID);
                detach_pid(p, PIDTYPE_SID);
        }

        REMOVE_LINKS(p);
        p->pid = 0;
}
```

first term

The first occurrence of a term, such as the first time we introduce a *bulletin-board* and note its abbreviated form, *bboard*.

## 6. Activate Your Subscription

Before you can access service and software maintenance information, and the support documentation included in your subscription, you must activate your subscription by registering with Red Hat. Registration includes these simple steps:

- Provide a Red Hat login
- Provide a subscription number
- Connect your system

The first time you boot your installation of Red Hat Enterprise Linux, you are prompted to register with Red Hat using the **Setup Agent**. If you follow the prompts during the **Setup Agent**, you can complete the registration steps and activate your subscription.

If you can not complete registration during the **Setup Agent** (which requires network access), you can alternatively complete the Red Hat registration process online at http://www.redhat.com/register/.

### 6.1. Provide a Red Hat Login

If you do not have an existing Red Hat login, you can create one when prompted during the **Setup Agent** or online at:

```
https://www.redhat.com/apps/activate/newlogin.html
```

A Red Hat login enables your access to:

- Software updates, errata and maintenance via Red Hat Network
- Red Hat technical support resources, documentation, and Knowledgebase

If you have forgotten your Red Hat login, you can search for your Red Hat login online at:

```
https://rhn.redhat.com/help/forgot_password.pxt
```

### 6.2. Provide Your Subscription Number

Your subscription number is located in the package that came with your order. If your package did not include a subscription number, your subscription was activated for you and you can skip this step.

You can provide your subscription number when prompted during the **Setup Agent** or by visiting http://www.redhat.com/register/.

### 6.3. Connect Your System

The Red Hat Network Registration Client helps you connect your system so that you can begin to get updates and perform systems management. There are three ways to connect:

1. During the **Setup Agent** — Check the **Send hardware information** and **Send system package list** options when prompted.
2. After the **Setup Agent** has been completed — From **Applications** (the main menu on the panel), go to **System Tools**, then select **Red Hat Network**.
3. After the **Setup Agent** has been completed — Enter the following command from the command line as the root user:
   - `/usr/bin/up2date --register`

## 7. More to Come

The *Red Hat SELinux Guide* is part of Red Hat's growing commitment to provide useful and timely support to Red Hat Enterprise Linux users. As new releases of Red Hat Enterprise Linux are made available, we make every effort to include both new and improved documentation for you.

### 7.1. Send in Your Feedback

If you spot a typo in the *SELinux Guide*, or if you have thought of a way to make this manual better, we would love to hear from you. Please submit a report in Bugzilla (http://bugzilla.redhat.com/bugzilla) against the component `rhel-selg`.

Be sure to mention the manual's identifier:

```
rhel-selg(EN)-4-Print-RHI (2005-02-15-T16:20)
```

If you mention this manual's identifier, we will know exactly which version of the guide you have.

If you have a suggestion for improving the documentation, try to be as specific as possible. If you have found an error, please include the section number and some of the surrounding text so we can find it easily.

# I. Understanding SELinux

This part provides an overview and theory of SELinux in general and policy in particular.

## Table of Contents

# Chapter 1.

# SELinux Architectural Overview

This chapter is an overview of the SELinux architecture, building upon what was discussed in Section 1 *What Is SELinux?*. The technical information you learn here helps you accomplish your goals in an SELinux environment. This chapter discusses the interaction of SELinux policy, the kernel, and the rest of the OS. Chapter 2 *SELinux Policy Overview* provides a more detailed look into the policy itself.

## 1.1. Flask Security Architecture and SELinux

For a history of SELinux and the Flask architecture, read Appendix A *Brief Background and History of SELinux*.

Flask was developed to work through some of the inherent problems with a MAC architecture. Traditional MAC is closely integrated with the *multi-level security* (MLS) model. Access decisions in MLS are based on clearances for subjects and classifications for objects, with the objective of *no read-up, no write-down* . This provides a very static lattice that allows the system to decide by a subject's security clearance level which objects can be read and written to. The focus of the MLS architecture is entirely on maintaining confidentiality.

The inflexible aspect of this kind of MAC is the focus on confidentiality. The MLS system does not care about integrity of data, least privilege, or separating processes and objects by their duty, and has no mechanisms for controlling these security needs. MLS is a mechanism for maintaining confidentiality of files on the system, by making sure that unauthorized users cannot read from or write to them.

Flask solves the inflexibility of MLS-based MAC by separating the policy enforcement from the policy logic, which is also known as the *security server*. In traditional Flask, the security server holds the security policy logic, handling the interpretation of *security contexts*. Security contexts or *labels* are the set of security attributes associated with a process or an object. Such security labels have the format of `<user>:<role>:<type>`, for example, `system_u:object_r:httpd_exec_t`. The SELinux user `system_u` is a standard identity used for daemons. The role `object_r` is the role for system objects such as files and devices. The type `httpd_exec_t` is the type applied to the `httpd` executable `/usr/sbin/httpd`. The label elements user, role, and type are explained in Section 2.10 *SELinux Users and Roles* and Section 2.7 *TE Rules - Types*.

Prior to full integration with the Linux kernel, security contexts were maintained separately in a file as a set of *security identifiers* or *SID*s. Part of the change when moving to the Linux 2.6.*x* kernel is the usage of extended attributes (EAs) in the file system. SIDs are not entirely retired, but they are no longer exported to userspace from the kernel. For example, the kernel has some initial SIDs used by `init` during bootstrapping before the policy is loaded. In addition, `libselinux` provides a userspace SID abstraction for applications that enforce policy, such as `dbus-daemon` and `nscd`. Otherwise, users and other programs only interact with security contexts. To minimize confusion, from here forward in this guide, the term *security context* is used to include the SID.

The security server need only do a look-up with a pair of contexts on a matrix of type-labeled subjects and objects, and the result is put in the *access vector cache* (*AVC*) for retrieval on subsequent matching requests.

By adding in a generalized form of TE that is separated into its own security subsystem, Flask can be flexible in labeling for transition and access decisions. Instead of being tied to a rigidly defined lattice of relationships, Flask can define other labels based on user identity (UID), role attributes, domain or type attributes, MLS levels, and so forth.

Similarly, access decision computations can be made using multiple methods in the same decision. These methods could be lattice models, static matrix lookups, historical decisions, environmental decisions, or policy logic obtained in real time. These computations are all handled by the policy engine and cached, leaving the policy enforcement code available to handle requests.

One other Flask flexibility is that any of these subsystems can be swapped out for a new or different system, and none of the other systems are even aware of the change. The abstraction between policy enforcement and policy decision-making is what makes this possible. This flexibility gives Red Hat Enterprise Linux developers the control they need to make the best architecture decisions without being tied to a particular subsystem.

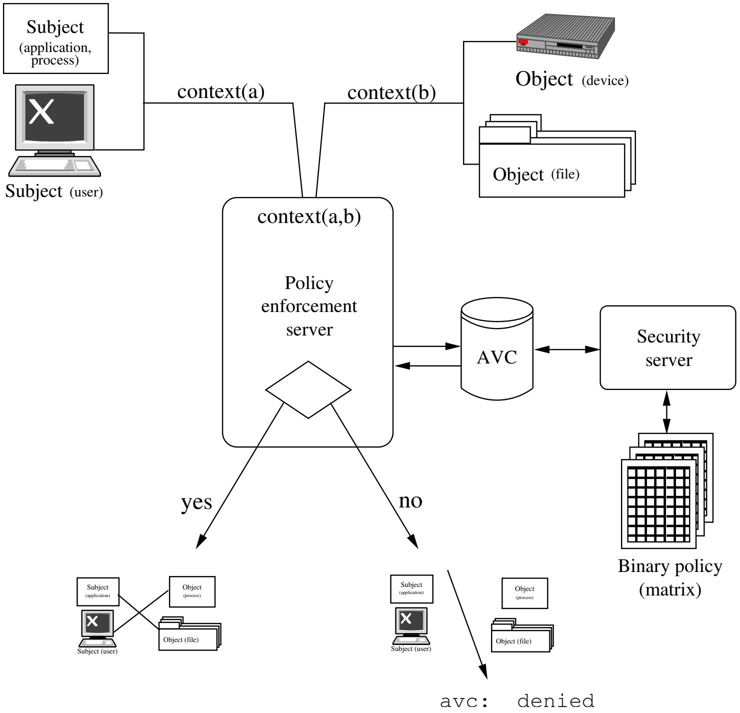


**Figure 1-1. Flask Architecture**

Figure 1-1 describes the Flask architecture, showing the process of an operation. In this operation, standard DAC has occurred, which means the subject already has gained access to the object via regular Linux file permissions based on the UID[1]. The operation can be anything: reading from or writing to a file/device, transitioning a process from one type to another type, opening a socket for an operation, delivering a signal call, and so forth.

1. A subject, which is a process, attempts to perform an operation on an object, such as a file, device, process, or socket.
2. The policy enforcement server gathers the security context from the subject and object, and sends the pair of labels to the security server, which is responsible for policy decision making.

[1] This type of access control is also called *identify-based access control* or *IBAC*.

3. The policy server first checks the AVC, and returns a decision to the enforcement server.

   If the AVC does not have a policy decision cached, it turns to the security server, which uses the binary policy that is loaded into the kernel during initialization. The AVC caches the decision, and returns the decision to the enforcement server, that is, the kernel.
4. If the policy permits the subject to perform the desired operation on the object, the operation is allowed to proceed.
5. If the policy does not permit the subject to perform the desired operation, the action is denied, and one or more `avc: denied` messages are logged to `$AUDIT_LOG`, which is typically `/var/log/messages` in Red Hat Enterprise Linux.

With the security server handling the policy decision making, the enforcement server handles the rest of the tasks. In this role, you can think of the enforcement code as being an *object manager*. Object management includes labeling objects with a security context, managing object labels in memory, and managing client and server labeling.

## 1.2. SELinux, an Implementation of Flask

SELinux has been through several iterations as part of the process of being incorporated into the Linux kernel. During this time, the overall architecture has remained the same, but many of the programmatic details have changed. Some of the reasons for change were: requirements for upstream acceptance; changes in LSM as part of being accepted into the kernel; and the switch to using xattrs.

As one example of the changes between kernel versions, originally security context was maintained through a mapping from context to SID, and managed by the security server. In the 2.6.*x* Linux kernel, the security context for a file is stored in the xattrs, allowing it to carry around its own SELinux context.

As an implementation of the Flask architecture, SELinux also served as a reference implementation of LSM. Originally LSM and SELinux were patches to the 2.4.<*x*> series of kernels; SELinux was never able to work as a loadable security module. Therefore, a big part of gaining upstream acceptance into the mainline Linux kernel required everything from fixing coding practices to changing how SELinux interacted with the kernel.

Part of the SELinux development team was also instrumental in designing, building, and integrating LSM into the kernel. SELinux integration into the kernel was the motivation to start the LSM project. SELinux was an early proof of the ability of LSM to allow security-enhancements to be connected into, instead of strapped onto, the Linux kernel. Originally, SELinux was a loadable module, but it became statically compiled into the 2.6.*x* kernel. It is still an LSM module, using the LSM hooks in the kernel to control and label. Because of the abstraction layer provided by both the LSM and Flask frameworks, SELinux is highly configurable and modifiable.

Flask is flexible enough to work in many different environments, and Linux is a natural fit for the Flask model. Access to the kernel source and a willing, community-driven development process allow for the best modification to fully support Flask's objectives. The wide range of platforms Linux runs on means SELinux is extensively tested. The consensus process of getting SELinux integrated into the kernel has improved the code and practices. Now that it is integrated, it has a better chance of long-term success than security-enhancement models that are strapped on-top of the operating system.

There are a few more differences in the specific way SELinux implements Flask in the Linux kernel, compared to traditional Flask methodology and initial SELinux creation:

1. Under traditional TE, there is a distinction between types and domains. A type is the security context for a file object, and a domain is the security context for a process. In the SELinux implementation, there is no real distinction programmatically. In SELinux, domains are processes that have the *attribute* `process`, so the term domain is used in the traditional way. Similarly, the term type is mostly applied to object types, but it can mean both domains and types.

2. The term security server is still used for the sake of clarity, but it is no longer a stand-alone service. The security server, the AVC, and the policy engine are now all parts of the kernel.

# Chapter 2.

# SELinux Policy Overview

This chapter is an overview of SELinux policy, some of its internals, and how it works. This chapter discusses the policy in a more general way, where Chapter 3 *Targeted Policy Overview* focuses on the details of the targeted policy as it ships in Red Hat Enterprise Linux. This chapter starts with a brief overview of what policy is and where it resides. Next, the role of SELinux during boot is discussed. This is followed by discussions on file security contexts, object classes and permissions, attributes, types, access vectors, macros, users and roles, constraints, and a brief discussion summarizing special kernel interfaces.

To see all of the details discussed in this chapter, you must make sure you have installed the policy source and binary packages for the targeted policy:

- `selinux-policy-targeted-sources-<version>`
- `selinux-policy-targeted-<version>`

**Important**

When you have the policy sources installed, `rpm` may assume that you have modified the policy and may not automatically load a newly installed policy. This occurs if you have ever loaded the policy from source, that is, run `make load`, `make reload`, or `make install`. New binary policy packages install `policy.<XY>` as, for example, `$SELINUX_POLICY/policy.18.rpmnew`.

If you have not modified the policy or want to use the binary policy package, you can `mv policy.18.rpmnew policy.18`, then `touch /.autorelabel` and reboot. If you have modified the policy and want to load your modifications, you must upgrade the policy source package and `make load`. Policy building is discussed in Chapter 7 *Compiling SELinux Policy*.

If you have only built the policy but never loaded it, that is, have only run `make policy`, you should not run into this situation. The binary policy installs cleanly, knowing that you are not running a custom policy.

Work is ongoing to improve package installation logic so the entire process is automated by `rpm`. Expect this to be included in a future update to Red Hat Enterprise Linux 4.

## 2.1. What Is Policy?

Policy is the set of rules that guide the SELinux security engine. It defines types for file objects and domains for processes, uses roles to limit the domains that can be entered, and has user identities to specify the roles that can be attained. A *domain* is what a type is called when it is applied to a process.

A type is a way of grouping together like items based on their fundamental security sameness. This doesn't necessarily have to do with the unique purpose of an application or the content of a document. For example, an object such as a file can have any type of content and be for any purpose, but if it belongs to a user and lives in that user's home directory, it is considered to be of a specific security type, `user_home_t`.

These object types gain their sameness because they are accessible in the same way by the same set of subjects. Similarly, processes tend to be of the same type if they have the same permissions as other subjects. In the targeted policy, programs that run in the `unconfined_t` domain have an executable with a type such as `sbin_t`. From an SELinux perspective, that means they are all equivalent in terms of what they can and cannot do on the system.

For example, the binary executable file object at /usr/bin/postgres has the type of postgresql_exec_t. All of the targeted daemons have their own *_exec_t type for their executable applications. In fact, the entire set of PostgreSQL executables such as createlang, pg_dump, and pg_restore have the same type, postgresql_exec_t, and they transition to the same domain, postgresql_t, upon execution.

The policy defines various rules that say how each domain may access each type. Only what is specifically allowed by the rules is permitted. By default every operation is denied and audited, meaning it is logged in $AUDIT_LOG, such as /var/log/messages. Policy is compiled into binary format for loading into the kernel security server, and as the security server hands out decisions, these are cached in the AVC for performance.

Policy can be administratively defined, either by modifying the existing files or adding local TE and file context files to the policy tree. Such a new policy can be loaded into the kernel in real time. Otherwise, the policy is loaded during boot by init, as explained in Section 2.3 *Policy Role in Boot*. Ultimately, every system operation is determined by the policy and the type labeling of the files.

**Important**

> After loading a new policy, it is recommended to restart any services that may have new or changed labeling. For the most part, this is only the targeted daemons, as listed in Section 3.1 *What is the Targeted Policy?*.

SELinux is an implementation of domain-type access control, with role-based limiting. The policy specifies the rules in that environment. It is written in a simple language created specifically for writing security policy. Policy writers use m4 macros to capture common sets of low-level rules. There are a number of m4 macros defined in the existing policy, which assist greatly in writing new policy. These rules are preprocessed into many additional rules as part of building policy.conf, which is compiled into the binary policy.

The files are divided into various categories in a policy tree at $SELINUX_SRC/. This is covered in Section 3.2 *Files and Directories of the Targeted Policy*. Access rights are divided differently among domains, and no domain is required to act as a master for all other domains. Entering and switching domains is controlled by the policy, through login programs, userspace programs such as newrole, or by requiring a new process execution in the new domain, called a *transition*.

## 2.2. Where is the Policy?

There are two components to the policy, the binary tree and the source tree. The binary tree comes from the selinux-policy-<*policyname*> package and supplies the binary policy file. Alternately, the binary policy can be built from source when the selinux-policy-<*policyname*>-sources package is installed. For Red Hat Enterprise Linux 4 the <*policyname*> is targeted. Directory conventions for this guide are explained in Section 3 *Conventions for SELinux Directories and Files*.

- /etc/selinux/targeted/ — this is the root folder for the targeted policy, and contains both the binary and source trees.
- /etc/selinux/targeted/policy/ — the binary policy file policy.<*XY*> is here. In this guide, the variable $SELINUX_POLICY/ is used for this directory.
- /etc/selinux/targeted/src/policy/ — this is the location of the policy source tree. For details about these sub-directories, read Section 3.2 *Files and Directories of the Targeted Policy*. In this guide, the variable $SELINUX_SRC/ is used for this directory.
- /etc/selinux/targeted/contexts/ — location of the security context information and configuration files, which are used during runtime by various applications. This directory contains:

  - *_context* and default_type — various contexts used by applications, such as the userhelper_context used by userhelper.
  - files/* — the file file_contexts contains the default contexts for the whole file system. This is what restorecon references when relabeling. The file media contains the default contexts for media devices such as the CD-ROM and floppy disk.
  - users/* — in the targeted policy, only the file root is in this directory. These files are used for determining context on login, which is system_r:unconfined_t for root.

- booleans — this is where the runtime Booleans are configured. This is the canonical configuration file when Boolean values are changed.

To help applications that need the various SELinux paths, libselinux has a number of functions that return the paths to the different configuration files and directories. This keeps applications from having to hard code the paths, especially since the active policy location is dependent on the setting in /etc/selinux/config. The list of functions is available from the manual page which you can view with the command man 3 selinux_binary_policy_path.

## 2.3. Policy Role in Boot

SELinux plays an important role early in system start-up. Since all of the processes must be labeled with their proper domain, init does some essential actions early in the boot process that keep labeling and policy enforcement in sync.

1. After the kernel has been loaded during boot, the initial process is assigned the predefined *initial SID* kernel. Initial SIDs are used for bootstrapping before the policy is loaded.
2. /sbin/init mounts /proc/, then looks for the selinuxfs file system type. If it is present, that means SELinux is enabled in the kernel.
3. If init does not find SELinux in the kernel, finds it is disabled via the *selinux=0* boot parameter, or if /etc/selinux/config specifies that SELINUX=disabled, boot proceeds with a non-SELinux system.

   At the same time, init sets the enforcing status if it is different from the setting in /etc/selinux/config. This happens when a parameter is passed during boot. The default mode is permissive until the policy is loaded, then enforcement is set by the configuration file or by the parameters *enforcing=0* or *enforcing=1*.
4. If SELinux is present, /selinux/ is mounted.
5. The kernel checks /selinux/policyvers for the supported policy version. init looks into /etc/selinux/config to see which policy is active, such as the targeted policy, and loads the associated file at $SELINUX_POLICY/policy.<*version*>.

   If the binary policy is *not* the version supported by the kernel, init attempts to load the policy file if it is a previous version. This provides backward compatibility with older policy versions.

   If the local settings in /etc/selinux/targeted/booleans are different from those compiled in the policy, init modifies the policy in memory based on the local settings prior to loading the policy into the kernel.
6. Now that the policy is loaded, the initial SIDs are mapped to security contexts in the policy, as defined in $SELINUX_SRC/initial_sid_contexts. In the case of the targeted policy, the new domain is user_u:system_r:unconfined_t. The kernel can now begin to get security contexts dynamically from the in-kernel security server.

7. `init` then re-executes itself so that it can transition to a different domain, if the policy defines it. For the targeted policy, there is no transition defined and `init` remains in the `unconfined_t` domain.

8. At this point, `init` continues with its normal boot.

The reason for `init` to re-execute itself is to accommodate stricter SELinux policy controls. The objective of a re-execution is to transition to a new domain with its own granular rules. The only way a process can gain a domain is during execution, meaning such programs are the only *entry points* into the domains. For example, if the policy has a specific domain for `init` such as `init_t`, there has to be a method to get from the initial SID, such as `kernel`, to the proper runtime domain for `init`. Because this transition may need to occur, `init` is coded to re-execute itself after loading the policy.

This transition with `init` happens if the rule `domain_auto_trans(kernel_t, init_exec_t, <target_domain_t>)` is present in the policy. This rule states that an automatic transition occurs on anything executing from the `kernel_t` domain that executes a file of type `init_exec_t`. When this execution occurs, the new process is assigned the domain <*`target_domain_t`*>, using an actual target domain such as `init_t`.

## 2.4. File System Security Contexts

This section covers how file system security contexts are defined and stored.

SELinux stores file security labels in xattrs[1]. For more information about xattrs, read the manual pages for `attr(5)`, `getfattr(1)`, and `setfattr(1)`. Xattrs are stored as name-value property pairs associated with files. SELinux uses the *`security.selinux`* attribute. The xattrs can be stored with files on a disk or in memory with pseudo file systems. Currently, most file system types support the API for xattr, which allows for retrieving attribute information with `getxattr(2)`.

Some non-persistent objects can be controlled through the API. The pseudo-tty system controlled through `/dev/pts` is manipulated through `setxattr(2)`, enabling programs such as `sshd` to change the context of a tty device. Information about the tty is exported and available through `getxattr(2)`. However, `libselinux` provides a more useful set of functions layered on top of the xattr API, such as `getfilecon(3)`, `setfilecon(3)`, and `setfscreatecon(3)`.

**Tip**

It is recommended to use `libselinux` when managing file attributes in SELinux programmatically.

There are two approaches to take for storing file security labels on a file system, such as ext2 or ext3. One approach is to label every file system object (all files) with an individual security attribute[2]. Once these labels are on the file system, the xattrs become authoritative for holding the state of security labels on the system.

The other option is to label the entire file system with a single security attribute. This is called *genfs labeling*. One example of this is with ISO9660 file systems, which are used for CD-ROMs and `.iso` files. This example from `$SELINUX_SRC/genfs_contexts` defines the context for every file on an ISO9660 file system.

[1] Extended attributes are also called *EAs*. To be more concise, the term xattr is used in this guide.

[2] These are defined initially for the system in `$SELINUX_SRC/file_contexts/types.fc`. This file uses regular expression matching to associate the files on a particular path with a particular security label. These contexts are rendered into the installed version at `/etc/selinux/targeted/contexts/files/file_contexts`, and are used during installation of the operating system and software packages, or for checking or restoring files to their original state.

```
genfscon iso9660 /                    system_u:object_r:iso9660_t
```

The file genfs_contexts has labels to associate with the most common mounted file systems that do not support xattrs.

You can set the context at the time of mounting the file system with the option -o context=<*user*>:<*role*>:<*type*>. A complete list of file system types can be found at $SELINUX_SRC/types/file.te. This option is also known as *mountpoint labeling* and is new in the 2.6.*x* kernel. Mountpoint labeling occurs in the kernel memory only, the labels are not written to disk. This example overrides the setting in genfs_contexts that would normally mount the file system as nfs_t:

```
mount -t nfs -o context=user_u:object_r:user_home_t \
<hostname>:/shares/homes/ /home/
```

The -o context= option is useful when mounting file systems that do not support extended attributes, such as a floppy or hard disk formatted with VFAT, or systems that are not normally running under SELinux, such as an ext3 formatted disk from a non-SELinux workstation. You can also use -o context= on file systems you do not trust, such as a floppy. It also helps in compatibility with xattr-supporting file systems on earlier 2.4.<*x*> kernel versions. Even where xattrs are supported, you can save time not having to label every file by assigning the entire disk one security context.

Two other options are -o fscontext= and -o defcontext=, both of which are mutually exclusive of the context option. This means you can use fscontext and defcontext with each other, but neither can be used with context.

The fscontext option works for all file systems, regardless of their xattr support. The fscontext option sets the overarching file system label to a specific security context. This file system label is separate from the individual labels on the files. It represents the entire file system for certain kinds of permission checks, such as during mount or file creation. Individual file labels are still obtained from the xattrs on the files themselves. The context option actually sets the aggregate context that fscontext provides, in addition to supplying the same label for individual files.

You can set the default security context for unlabeled files using defcontext. This overrides the value set for unlabeled files in the policy and requires a file system that supports xattr labeling. This example might be for a shared volume that gets file drops of security quarantined code, so the dropped files are labeled as being unsafe and can be controlled specially by policy:

```
mount -t ext3 defcontext=user_u:object_r:insecure_t \
  /shares/quarantined
```

This all works because SELinux acts as a transparent layer for the mounted file system. After parsing the security options, SELinux only passes normal file system specific code to the mounted file system. SELinux is able to seamlessly handle the text name-value pairs that most file systems use for mount options. File systems with binary mount option data, such as NFS and SMBFS, need to be handled as special cases. Currently, NFSv3 is the only one supported.

## 2.4.1. Security Contexts and the Kernel

SELinux uses LSM hooks in the kernel in key locations, where they interject access vector decisions. For example, there is a hook just prior to a file being read by a user, where SELinux steps from the normal kernel workflow to request the AVC decision. This mainly occurs between a subject (a process such as less) and an object (a file such as /etc/ssh/sshd_config) for a specific permission need (such as read).

Based on the result read back from the AVC, the hook either continues the workflow or returns EACCES, that is, Permission denied.

The way SELinux implements its label in the xattr is different from other labeling schemes. SELinux stores its labels in human-readable strings. This provides a meaningful label with the file that can help in backup, restoration, and moving files between systems. Standard attributes do not provide a label that has continuous meaning for the file.

In this example under the targeted policy, the policy does not specify anything about files created by `unconfined_t` in the directory `/tmp`, so the files inherit the context from the parent directory:

```
id -Z
root:system_r:unconfined_t
ls -dZ /tmp
drwxrwxrwt  root   root    system_u:object_r:tmp_t       /tmp/
touch /tmp/foo
ls -Z /tmp/foo
-rw-r--r--  root   root    root:object_r:tmp_t           /tmp/foo
```

In this example under a different policy, the policy explicitly states that files created by `user_t` in `/tmp` have a type of `user_tmp_t`:

```
id -Z
user_u:staff_r:user_t
ls -dZ /tmp
drwxrwxrwt  usera  usera   system_u:object_r:tmp_t       /tmp/
touch /tmp/foo
ls -Z /tmp/foo
-rw-r--r--  usera  usera   root:object_r:user_tmp_t      /tmp/foo
```

This finer grained control is implemented via policy using the `tmp_domain()` macro, which defines a temporary type per domain. In this macro, the variable `$1_tmp_t` is expanded by substituting the subject's type base, so that `user_t` creates files with a type of `user_tmp_t`.

Having separate types for `/tmp/` protects a domain's temporary files against tampering or disclosure by other domains. It also protects against misdirection through a malicious symlink. In the targeted policy, the confined daemons have separate types for their temporary files, keeping those daemons from interfering with other `/tmp/` files.

A privileged application can override any stated labeling rule by writing a security context to `/proc/self/attr/fscreate` using `setfscreatecon(3)`. This action must still be allowed by policy. The context is then used to label the next newly created file object, and the `fscreate` is automatically reset after the next `execve` or through `setfscreatecon(NULL)`. This ensures that a program starts in a known state without having to be concerned what context was left by the previous program in `/proc/self/attr/fscreate`.

## 2.5. Object Classes and Permissions

SELinux defines a number of classes for objects, making it easier to group certain permissions by specific classes. Here are some examples:

- File related classes include `filesystem` for file systems, `file` for files, and `dir` for directories. Each class has it's own associated set of permissions. The `filesystem` class can mount, unmount, get attributes, set quotas, relabel, and so forth. The `file` class gains the common file permissions such as read, write, get and set attributes, lock, relabel, link, rename, append, etc.
- Network related classes include `tcp_socket` for TCP sockets, `netif` for network interfaces, and `node` for network nodes. The `netif` class, for example, can send and receive on TCP, UDP and raw sockets (`tcp_recv`, `tcp_send`, `udp_send`, `udp_recv`, `rawip_recv`, and `rawip_send`.)

The object classes have matching declarations in the kernel, meaning that it is not trivial to add or change object class details. The same thing is true for permissions. Development work is ongoing to make it possible to register and unregister classes and permissions dynamically.

Permissions are the actions that a subject can take on an object, if the policy allows it. These permissions are the access requests that SELinux actively allows or denies.

There are several common sets of permissions defined in the targeted policy, in `$SELINUX_SRC/flask/access_vectors`. These allow the actual classes to inherit the sets, instead of rewriting the same permissions across multiple classes:

```
# Define a common prefix for file access vectors.
#

common file
{
 ioctl
 read
 write
 create
 getattr
 setattr
 lock
 relabelfrom
 relabelto
 append
 unlink
 link
 rename
 execute
 swapon
 quotaon
 mounton
}

# Define a common prefix for socket access vectors.
#

common socket
{
# inherited from file
 ioctl
 read
 write
 create
 getattr
 setattr
 lock
 relabelfrom
 relabelto
 append
# socket-specific
 bind
 connect
 listen
 accept
 getopt
 setopt
 shutdown
 recvfrom
 sendto
 recv_msg
 send_msg
 name_bind
}

# Define a common prefix for ipc access vectors.
#

common ipc
{
 create
 destroy
 getattr
 setattr
 read
 write
 associate
 unix_read
 unix_write
}
```

Following the common sets are all the access vector definitions. The definition is structured this way: `class <class_name> [ inherits <common_name> ] { <permission_name> ... }`. A good example is the `dir` class, which inherits the permissions from the `file` class, and has additional permissions on top:

```
class dir
inherits file
{
 add_name
 remove_name
 reparent
 search
 rmdir
}
```

Another example is the class for `tcp_socket`, which inherits the `socket` set plus having its own set of additional permissions:

```
class tcp_socket
inherits socket
{
 connectto
 newconn
 acceptfrom
 node_bind
}
```

## 2.6. TE Rules - Attributes

Policy attributes identify as groups sets of security types that have a similar property. These groups can be controlled by fewer, overarching rules. The relationship is many-to-many: a type can have any amount of attributes, and an attribute can be associated with any number of types.

The declarations file `$SELINUX_SRC/attrib.te` is well documented in the comment blocks. The attribute declaration syntax is: `attribute <identifier>`:

```
## Samples from $SELINUX_SRC/attrib.te

# The domain attribute identifies every type that can be
# assigned to a process.  This attribute is used in TE rules
# that should be applied to all domains, e.g. permitting
# init to kill all processes.
attribute domain;

# Identifies all default types assigned to packets received
# on network interfaces.
attribute netmsg_type;
```

Here are a few noteworthy attributes. Information about these was obtained through policy analysis using **apol**, part of the `setools` package. You can read more about this in Section 6.3 *Using **apol** for Policy Analysis.*:

`httpdcontent`

The purpose of this attribute is to group together the various types associated with the policy for Apache HTTP. Because of the complexity of the `httpd` configuration, the targeted policy includes Boolean values that allow you to grant blanket permissions for `httpd` content types. This helps Web applications and built-in scripting, such as PHP for Apache HTTP, to work with the content. The types in this attribute are:

```
# This is an aliasing relationship
httpd_sys_content_t: httpd_sysadm_content_t, \
  httpd_user_content_t

# These types handle different permissions sets for scripts
httpd_sys_script_ro_t
httpd_sys_script_rw_t
httpd_sys_script_ra_t
```

The first line in the attribute group specifies that `httpd_sys_content_t` is an alias for `httpd_sysadm_content_t` and `httpd_user_content_t`.

`file_type`

This attribute is for all the types that are assigned to files, allowing for easier association of all file types to various kinds of file system needs. This attribute makes it more convenient to allow specific domains access to all file types. The list of types associated with the `file_type` attribute is greater than 170 types:

```
...
device_t
xconsole_device_t
file_t
default_t
root_t
mnt_t
home_root_t
lost_found_t
boot_t
system_map_t
boot_runtime_t
tmp_t
etc_t: hotplug_etc_t
shadow_t
ld_so_cache_t
etc_runtime_t
fonts_t
etc_aliases_t
net_conf_t: resolv_conf_t
lib_t
shlib_t
...
```

`netif_type`, `port_type`, and `node_type`

These attributes relate to network activity by domains. The `netif_type` identifies the types associated with network interfaces, allowing policy to control sending, receiving, and various operations on the interface:

```
netif_t
netif_eth0_t
netif_eth1_t
netif_eth2_t
netif_lo_t
netif_ippp0_t
netif_ipsec0_t
netif_ipsec1_t
netif_ipsec2_t
```

The `port_type` attribute is associated with all types that are assigned to port numbers. This allows SELinux to control port binding, meaning daemons are restricted in using a port depending on the type assigned to the port:

```
dns_port_t
dhcpd_port_t
http_cache_port_t
port_t
reserved_port_t
http_port_t
pxe_port_t
smtp_port_t
mysqld_port_t
rndc_port_t
ntp_port_t
portmap_port_t
postgresql_port_t
snmp_port_t
syslogd_port_t
```

The `node_type` is for types assigned to network nodes or hosts, allowing SELinux to control traffic to and from the node:

```
node_t
node_lo_t
node_internal_t
node_inaddr_any_t
node_unspec_t
node_link_local_t
node_site_local_t
node_multicast_t
node_mapped_ipv4_t
node_compat_ipv4_t
```

`fs_type`

This attribute identifies all types assigned to file systems, including non-persistent file systems. The `fs_type` attribute is used in TE rules to allow most domains to obtain overall file system statistics, and for some specific domains to mount any file system. Here are the SELinux file types that are part of `fs_type`:

```
devpts_t: sysadm_devpts_t, staff_devpts_t, user_devpts_t
fs_t
eventpollfs_t
```

```
futexfs_t
bdev_t
usbfs_t
nfsd_fs_t
rpc_pipefs_t
binfmt_misc_fs_t
tmpfs_t
autofs_t
usbdevfs_t
sysfs_t
iso9660_t
romfs_t
ramfs_t
dosfs_t
cifs_t: sambafs_t
nfs_t
proc_t
security_t
```

`exec_type`

This attribute groups together all types that are assigned to entry point executables. Any TE rules and assertions that should be applied to all entry point executables use this attribute. Here are the domains in this attribute:

```
ls_exec_t
shell_exec_t
httpd_exec_t
httpd_suexec_exec_t
httpd_php_exec_t
httpd_helper_exec_t
dhcpd_exec_t
hotplug_exec_t
initrc_exec_t
run_init_exec_t
init_exec_t
ldconfig_exec_t
mailman_queue_exec_t
mailman_mail_exec_t
mailman_cgi_exec_t
depmod_exec_t
insmod_exec_t
update_modules_exec_t
sendmail_exec_t
mysqld_exec_t
named_exec_t
ndc_exec_t
nscd_exec_t
ntpd_exec_t
ntpdate_exec_t
portmap_exec_t
postgresql_exec_t
rpm_exec_t
snmpd_exec_t
squid_exec_t
syslogd_exec_t
udev_exec_t
udev_helper_exec_t
winbind_exec_t
ypbind_exec_t
```

mta_delivery_agent

This attribute allows for flexibility in choosing a *mail transfer agent* (MTA) such as sendmail or postfix. Rules allow it to perform mail handling and take tasks from mailman. However, this attribute is not used in the targeted policy since none of the MTAs are targeted daemons for Red Hat Enterprise Linux 4.

domain

This attribute is for all types that can be assigned to a process. This is the method for identifying what is a domain in SELinux. In other Type Enforcement systems, domains may be implemented separately from types. In SELinux, domains are essentially types with the domain attribute.

This attribute allows you to have rules that can be applied to all domains, such as allowing init to send signals to all processes. Another example is the following rule that allows all processes to perform a search on directory objects that have a type of var_t or var_run_t, that is, the directories /var and /var/run:

```
allow domain { var_run_t var_t } : dir search ;
```

Here are the domains covered by this attribute:

```
unconfined_t: kernel_t, init_t, initrc_t, sysadm_t, rpm_t, \
              rpm_script_t, logrotate_t
mount_t
httpd_t
httpd_sys_script_t
httpd_suexec_t
httpd_php_t
httpd_helper_t
dhcpd_t
ldconfig_t
mailman_queue_t
mailman_mail_t
mailman_cgi_t
system_mail_t
mysqld_t
named_t
ndc_t
nscd_t
ntpd_t
portmap_t
postgresql_t
snmpd_t
squid_t
syslogd_t
winbind_t
ypbind_t
```

reserved_port_type

This attribute identifies all the types that are assigned to any of the reserved network ports, that is, ports numbered lower than 1024. The attribute is used to control binding. An example binding rule is followed here by the types that are part of this attribute:

```
# The allow rule permits the domain portmap_t to bind to a
# port with a type of portmap_port_t, which is one of the
# types identified by the reserved_port_type attribute.  The
# dontaudit rule tells SELinux to never audit the access of
# portmap_t to a reserved_port_type.

allow portmap_t portmap_port_t:{ udp_socket tcp_socket } \
  name_bind;
dontaudit portmap_t reserved_port_type:tcp_socket name_bind;
# Types associated with the reserved_port_type attribute
```

```
http_port_t
smtp_port_t
rndc_port_t
ntp_port_t
portmap_port_t
snmp_port_t
syslogd_port_t
```

## 2.7. TE Rules - Types

SELinux uses types in various ways. After they are declared, they can be used to make rules for the transition decision process, type changing process, and access vector decisions and assertions.

**Note**

> Defining the type transitions does not enable them. By default, access is denied until specifically allowed.

Domains are types applied to processes, identified by the type having the `domain` attribute. The same type is used for the process itself and the associated `/proc` file. Typically, you see the domain used as the source context for system operations, that is, the domain is the doer. A domain can be a target context, such as when `init` is sending process signals to a daemon.

With every SELinux transaction involving at least one domain, the number and kind of domains is central to the complexity of the security policy. More domains means finer security control, with a matching increase in configuration and maintenance difficulties.

Type Declaration

This syntax defines how types are declared. A type must be declared before rules can be written about it. The targeted daemons have their top-level domain declared through the macro `daemon_domain()`, which is discussed in Section 3.4 *Common Macros in the Targeted Policy*.

```
## Syntax of a type declaration

type <typename> [aliases] [attributes];

## Examples

type httpd_config_t, file_type, sysadmfile;
# httpd_config_t is a system administration file

type http_port_t, port_type, reserved_port_type;
# httpd_port_t is a reserved port, that is, number less than 1024

type httpd_php_exec_t, file_type, sysadmfile, exec_type;
# httpd_php_exec_t is a sysadmin file that is an entry point
# executable
```

Type Transitions

A type transition results in a new process running in a new domain different from the executing process, or a new object being labeled with a type different from the source doing the labeling.

The rules define what domain and file type transitions occur by default. The domain transition default can be overridden if the process explicitly requests a particular context. File transition default is actually inherit-from-parent, that is, the new file receives its context from the parent directory unless an explicit transition rule makes it inherit-from-creator. For example, the directory `~/` has a type of `user_home_dir_t`, and policy specifies that files created in a directory with that type are labeled with `user_home_t`.

Transitions are defined through macros that combine the `type_transition` rule with a set of allow rules. The allow rules are macros with variables that support common transitioning needs. For more information about macros, refer to Section 2.9 *Policy Macros*.

```
## General syntax of a transition

type_transition <source_type(s)> <target_type(s)> : \
  <class(es)> <new_type>

# Note that all excepting the <new_type> can be
# multiple types and classes, surrounded by brackets { }


## Domain transition syntax

type_transition <current_domain> <type_of_program> : \
  process <new_domain>

# note that the object class is fixed to the process attribute


## Domain transition examples

type_transition httpd_t httpd_sys_script_exec_t:process \
  httpd_sys_script_t;

# When the httpd daemon running in the domain httpd_t executes
# a program of the type httpd_sys_script_exec_t, such as a CGI
# script, the new process is given the domain of
# httpd_sys_script_t


type_transition initrc_t squid_exec_t:process squid_t;

# When init exec()s a program of the type squid_exec_t, the new
# process is transitioned to the squid_t domain


## New object labeling syntax

type_transition <creating_domain> <parent_object_type> : \
  <class(es)> <new_type>

# Note that multiple classes are allowed using the
# { } brackets


## New object labeling example
```

```
type_transition named_t var_run_t:sock_file named_var_run_t;

# When a process in the domain named_t creates a socket file
# in a directory of the type var_run_t, the socket file is
# given the type named_var_run_t.  The directory with the
# type var_run_t is defined in the policy as /var/run/.
#
# This rule is only found in this format in policy.conf,
# and it is derived from the following rule in
# $SELINUX_SRC/domain/programs/named.te:

file_type_auto_trans(named_t, var_run_t, named_var_run_t, \
  sock_file)

# This rule evokes the file_type_auto_trans macro from
# $SELINUX_SRC/macros/core_macros.te, ultimately feeding the 4
# variables in to the macro file_type_trans($1,$2,$3,$4)
```

Type Changes

This kind of transition is not used in the targeted policy in Red Hat Enterprise Linux 4. Type changes are used by trusted applications to change the labels of objects, such as `login` relabeling the tty for a user session. For more information about type changes, refer to the sources found in Chapter 9 *References*.

## 2.8. TE Rules - Access Vectors

*Access vectors* (*AVs*) are the rules that allow domains to access various system objects. An AV is a set of permissions. A basic AV rule is a subject and object pair of types, a class definition for the object, and a permission for the subject. There is a general rule syntax that covers all the kinds of AV rules:

```
<av_kind> <source_type(s)> <target_type(s)>:<class(es)> \
<permission(s)>
```

All AV rules are considered by the policy enforcement engine as two types, one class, and one access permission. However, rules are written using attributes, sets, and macros to be more efficient. AV rules are simplified during policy compilation.

The parts of the AV rule are defined elsewhere in this chapter. This section describes the kinds of access vectors used in the AV rule at `av_kind`. `av_kind` is one of three rule types:

- `allow` — permit a subject to act in a specific way with an object. The rule here allows `named` (in the domain of `named_t`) to perform a search of a directory with the type `sbin_t` (for example, `/sbin`, `/usr/sbin`, `/opt/sbin`, etc.):
  ```
  allow named_t sbin_t:dir search;
  ```
  If the ruling results in a denial, the denial is audited (that is, logged). Granted permission events are not logged.
- `auditallow` — when the permission is granted, log the access decision. In the targeted policy, there is only one `auditallow` rule. This rule logs usage of certain SELinux applications, for example logging `avc: granted { setenforce }` when allowing `setenforce`:
  ```
  auditallow unconfined_t security_t : security { load_policy \
    setenforce setbool };
  ```
- `dontaudit` — never audit a specific access denial. This is used when a program is attempting an action that is not allowed by policy, and the resulting denials are filling the log, but the denial is

not affecting the application doing its tasks. This AV lets you silently deny and ignore the access violation. For example, this `dontaudit` rule says to ignore when the `named_t` domain attempts to read or get attributes on a file with the `root_t` type. Denial of this access attempt does not effect `named` doing its job, so the denial is ignored to keep the logs clean:

```
dontaudit named_t root_t:file { getattr read };
```

There is one additional AV rule, `neverallow`. This AV assertion, defined in `$SELINUX_SRC/assert.te`, is not part of the regular permission checking. The purpose of this rule is to declare access vectors that must never be allowed. These are used to protect against policy writing mistakes, especially where macros can provide unexpected rights. These assertions are checked by the policy compiler, `checkpolicy`, when the policy is built, but after the entire policy has been evaluated, and are not part of the runtime access vector cache.

Here is the syntax and an example. In practice, a wildcard character `*` is often used to cover all instances possible in a rule field. The syntax is different in that it is possible to use `ifdef()` statements as sources or targets:

```
# Syntax for AV assertion

neverallow <source_name(s)> <target_name(s)> : \
  <class(es)> <permission(s)>
```

In this example from `assert.te`, the `neverallow` rule verifies that every type that a domain can enter into has the attribute `domain`. This prevents a rule from elsewhere in the policy allowing a domain to transition to a type that is not a process type. The tilde in front, `~domain`, means "anything that is not a domain":

```
# Verify that every type that can be entered by
# a domain is also tagged as a domain.
#
neverallow domain ~domain:process transition;
```

## 2.8.1. Understanding an `avc: denied` Message

When SELinux disallows an operation, a denial message is generated for the audit logs. In Red Hat Enterprise Linux, `$AUDIT_LOG` is `/var/log/messages`. This section explains the format of these log messages. For suggestions on using an `avc: denied` message for troubleshooting, refer to Section 5.2.11 *Troubleshoot User Problems With SELinux*.

Example 2-1 shows a denial generated when a user's Web content residing in `~/public_html` does not have the correct label.

```
Jan 14 19:10:04 hostname kernel: audit(1105758604.519:420):  \
avc:  denied  { getattr } for  pid=5962 exe=/usr/sbin/httpd \
path=/home/auser/public_html dev=hdb2 ino=921135 \
scontext=root:system_r:httpd_t \
tcontext=user_u:object_r:user_home_t tclass=dir
```

**Example 2-1. AVC Denial Message**

This shows the message parts and an explanation of what the part means:

### `avc: denied` Message Explained

`Jan 14 19:10:04`

Timestamp on the audit message.

`hostname`

The hostname of the system.

`kernel: audit(1105758604.519:420):`

This is the kernel audit log message pointer. The timestamp consists of a long number, which is the unformatted current time, and a short number, which is the milliseconds, that is, <*current_time*>.<*milliseconds_past_current_time*>. The third number is the serial number, which helps in stitching together the full audit trail from multiple messages. Multiple messages for the same event occur when full audit logging is enabled using an audit daemon, which logs various kernel events.

`avc: denied`

The operation was denied. A few operations have `auditallow` set so they generate `granted` messages instead.

`{ getattr }`

What was denied or granted. The brackets `{}` contain the actual permission that was attempted.

`for pid=5962`

The process ID of the application that is the source of the operation.

`exe=/usr/sbin/httpd`

The application being denied.

`path=/home/auser/public_html`

The path to the target file or directory the operation was attempted on.

`dev=hdb2`

The device node that holds the file system. The object of the denied operation lives in this file system.

`ino=921135`

The inode number of the target file or directory.

`scontext=root:system_r:httpd_t`

The security context of the source, that is, the process being denied access.

`tcontext=user_u:object_r:user_home_t`

The security context of the target, that is, the file or directory that is denied.

`tclass=dir`

The object class of the target, indicating that it was the directory `/home/auser/public_html/` that was being blocked.

## 2.9. Policy Macros

Macros are used throughout programming, as they provide reusable pieces of code that you can call one time and have explode into many meaningful lines. SELinux uses the `m4` macro language for writing reusable policy rules. This makes policy writing and management easier. In using macros, policy writers gain flexibility, modularity, shared quality control, and central management for complex pieces of policy.

Macros do not exist in the `policy.conf` file, as that file represents the exploded macro policy code. It is possible to work backward in finding where a particular `policy.conf` entry exists. If a daemon has a rule that you cannot find in the associated TE file at `$SELINUX_SRC/domains/program/<foo>.te`, then it is likely to be found in the macros. This section first explains the syntax and usage of a macro, then discusses the analysis method in more detail.

You can find more resources about `m4` from the manual page `man m4`, installed documentation at `/usr/share/doc/m4-<version>`, and through the resources listed in Chapter 9 *References*. Some of the specific macros used in the targeted policy are explained in Section 3.4 *Common Macros in the Targeted Policy*.

This usage example shows the first few lines from the Apache HTTP macro file, `$SELINUX_SRC/macros/program/apache_macros.te`:

```
define('apache_domain', '

#This type is for webpages
#
type httpd_$1_content_t, file_type, homedirfile, httpdcontent, \
  sysadmfile;
ifelse($1, sys, '
typealias httpd_sys_content_t alias httpd_sysadm_content_t;
')

# This type is used for .htaccess files
#
type httpd_$1_htaccess_t, file_type, sysadmfile;

...
```

The `define('apache_domain','` is the beginning of the macro definition. Inside the definition, the `$1` represents the parameter passed to the macro. Look in `$SELINUX_SRC/domains/program/apache.te`, which has the following invocation:

```
apache_domain(sys)
```

This single line then generates a large set of types and rules, substituting the passed parameter `sys` for every `$1`:

```
type httpd_$1_htaccess_t, file_type, sysadmfile; -> \
  type httpd_sys_htaccess_t, file_type, sysadmfile;
type httpd_$1_script_exec_t, file_type, sysadmfile -> \
  type httpd_sys_script_exec_t, file_type, sysadmfile
role system_r types httpd_$1_script_t; -> \
  role system_r types httpd_sys_script_t;
...
```

## 2.9.1. How To Backtrack a Rule

To find how a rule is derived from a macro, follow this approach. Take a rule you are curious about:

```
allow httpd_t httpd_suexec_t:process transition;
...
type_transition httpd_t httpd_suexec_exec_t:process httpd_suexec_t;
```

The allow rule says, httpd_t is permitted to start a child process that transitions to httpd_suexec_t. The type_transition rule defines two things: the circumstances, that is, when the domain httpd_t is executing a file of the type httpd_suexec_exec_t (/usr/sbin/suexec); and the child domain transitioned to, httpd_suexec_t. The allow rule then permits the defined transition.

These rules are present only in $SELINUX_SRC/policy.conf, so they must be derived from a macro.

In those rules, the variable elements are the parent domain (httpd_t), the child domain (httpd_suexec_t), and the program type (httpd_suexec_exec_t). These are represented in the macro as $1, $2, and so forth. Fortunately, the search is made easier because the object class (process) and permission (transition) are never variables in an SELinux macro. It is safe to search using the class and permission as a query:

```
grep -R ":process transition" macros/*
macros/core_macros.te:allow $1 $3:process transition;    # match
macros/global_macros.te:allow $1 self:process transition;
```

The return from core_macros.te fits the right format. Here it is in the macro file, showing it to be part of the domain_trans() macro:

```
# domain_trans(parent_domain, program_type, child_domain)
#
# Permissions for transitioning to a new domain.
#

define('domain_trans','

#
# Allow the process to transition to the new domain.
#
allow $1 $3:process transition;
```

In the macro call, the variables are domain_trans($1, $2, $3), with $1 the parent domain, $2 the program type, and $3 the child domain.

However, a search through $SELINUX_SRC/domains/program/apache.te and $SELINUX_SRC/macros/programs/apache_macros.te does not find a line such as domain_trans(httpd_t, httpd_suexec_t, httpd_suexec_exec_t). This means that domain_trans() is not called directly by the Apache HTTP policy, so another macro must be involved.

Looking back at the rules you are curious about, the common name roots that make up those rules are httpd_t and httpd_suexec. Focusing your search on those as variables turns up a macro call:

```
grep httpd_suexec domains/program/apache.te  | grep httpd_t
daemon_sub_domain(httpd_t, httpd_suexec)
```

The parameter httpd_suexec does not have either of the suffixes, _t or _exec_t, because it obtains those from the macro. The macro daemon_sub_domain() is found in $SELINUX_SRC/macros/global_macros.te. Notice the _exec_t and _t that are attached to the variable inputs $1 and $2:

```
# define a sub-domain, $1_t is the parent domain, $2 is the name
# of the sub-domain.
#
define('daemon_sub_domain', '

...
```

```
domain_auto_trans($1, $2_exec_t, $2_t)
```

Recall that the variables fed into `daemon_sub_domain()` were `httpd_t` (`$1`) and `httpd_suexec` (`$2`). When `m4` runs, it inputs the parameters in the order received, so `$1` becomes `httpd_t`, `$2_exec_t` becomes `httpd_suexec_exec_t`, and `$2_t` is `httpd_suexec_t`. Notice that the macro `daemon_sub_domain` actually calls `domain_auto_trans()`, which is found in `core_macros.te` and looks like this:

```
define(‘domain_auto_trans’,‘
domain_trans($1,$2,$3)
type_transition $1 $2:process $3;
’)

...

define(‘domain_trans’,‘
allow $1 $3:process transition;
...
```

There you see the completion of the chain, as `domain_trans()` is called, and the parameters are fed in to create the rules you are looking for:

```
$1 = httpd_t             (base input of httpd_t)
$2 = httpd_suexec_exec_t (base input of httpd_suexec)
$3 = httpd_suexec_t      (base input of httpd_suexec)

apache.te                                # feeds 2 variables into
 daemon_sub_domain(httpd_t, httpd_suexec)# which calls
  domain_auto_trans($1, $2_exec_t, $2_t) # that associates new vars
#### $1 = $1, $2_exec_t = $2, $2_t = $3) # and feeds the vars into
     domain_trans($1,$2,$3)              # which has
       type_transition $1 $2:process $3; # that expands into

type_transition httpd_t httpd_suexec_exec_t:process httpd_suexec_t
                                         # and
                                         # expands domain_trans()
        allow $1 $3:process transition;  # which expands into

allow httpd_t httpd_suexec_t:process transition;

# Here is a final association of variables to sources

allow    $1           $3    :process transition;
allow httpd_t httpd_suexec_t:process transition;

type_transition    $1           $2         :process       $3;
type_transition httpd_t httpd_suexec_exec_t:process httpd_suexec_t;
```

## 2.10. SELinux Users and Roles

**Important**

Users and roles can play a part in an SELinux policy. However, the greater part of SELinux is Type Enforcement. Additionally, the targeted policy is designed not to utilize users and roles. Every domain in the targeted policy runs in a single role, and TE is used to separate the confined processes from the other processes.

This is why, when discussing interaction of processes and files, the type component is what you focus on.

### 2.10.1. SELinux Roles

Roles define which SELinux user identities can have access to what domains. Roles are created by the existence of one or more declarations in a TE rules file in `$SELINUX_SRC/domains/*`:

Simply being in a role is not enough to allow domain transition. If a process is in `role_r` and `domain1_t`, and `role_r` is authorized for `domain2_t`, the process cannot jump to `domain2_t`. There must be an allow rule for the process transition to take place.

For example, the domains `named_t` and `squid_t` are both in the role `system_r`. However, `named_t` cannot transition to `squid_t` without an allow rule.

This shows the syntax and an allow example for role:

```
# syntax for role declaration

role <rolename> types <domain(s)>;

# example of role declaration from
# $SELINUX_SRC/domains/programs/ldconfig.te

role sysadm_r types ldconfig_t;
/* administrators are allowed access to ldconfig_t domain */


# syntax for role allow

allow <current_role(s)> <new_role(s)>;

# example of role allow from $SELINUX_SRC/rbac

allow user_r sysadm_r;
```

Grouping domains into roles is relatively intuitive, since roles are task-oriented[3]. Every process has a role, starting with the system role, `system_r`. Users gain a role at login, which includes using `su` where a new role can come with the new UID, or you can keep your UID and change your role using `newrole`; however, this is not often done. Domains are changed often and quietly, but roles rarely change, especially under the targeted policy. Another way to change roles is through a `role_transition`, this is also very rare and currently only used for an administrative role to launch daemons in a different role under a stricter policy:

```
role_transition sysadm_r $1_exec_t system_r;
/* When an administrator executes a process with the type of */
/* $1_exec_t, the process transitions from sysadm_r to       */
/* system_r.                                                 */
```

[3] Roles can encompass other roles, inheriting the privileges of the included role. This dominance capability is not used in the targeted policy. This syntax example shows a role that is declared to dominate over, and inherit the privileges of, `sysadm_r` and `user_r`:

```
dominance {role master_r {role sysadm_r; role user_r;} }
```

### 2.10.2. SELinux Users

SELinux user identities are different from UNIX identities. They are applied as part of the security label and can be changed in real time under limited conditions. SELinux identities are not primarily used in the targeted policy. In the targeted policy, processes and objects are system_u, and the default for Linux users is user_u. When identities are part of the policy scheme, they are usually identical to the Linux account name (UID), and are compiled into the policy. In such a strict policy, some system accounts may run under a generic, unprivileged user_u identity, while other accounts have direct identities in the policy database.[4]

For a review of the roles and users, including a discussion of the $SELINUX_SRC/users file, refer to Section 3.5 *Understanding the Roles and Users in the Targeted Policy*.

## 2.11. TE Rules - Constraints

These rules are defined in $SELINUX_SRC/constraints, and provide final and overarching constraints on the use of permissions that are enforced during runtime by the kernel security server. The constraints are in the form of Boolean expressions. The expression must be satisfied for the given permission to be granted.

For example, the following constraint pertains to a process transition. It says that when a transition takes place, the user identity on the process must remain the same through the transition. If httpd_t tries to transition to httpd_suexec_t, the user identity user_u must remain the same. The exception is if the source domain has the attribute privuser. It then has the privilege to change user identity:

```
constrain process transition ( u1 == u2 or t1 == privuser );
```

A constraint can make a restriction for the source and target based on type, role, or user identity. This is different from the other rule types. TE rules use only types, while role allow rules use a pair of roles.

This is from the constraints file and further explains syntax and constraints in the targeted policy:

```
# Define the constraints
#
# constrain class_set perm_set expression ;
#
# expression : ( expression )
#       | not expression
#       | expression and expression
#       | expression or expression
#       | u1 op u2
#       | r1 role_op r2
#       | t1 op t2
#       | u1 op names
#       | u2 op names
#       | r1 op names
#       | r2 op names
#       | t1 op names
#       | t2 op names
#
# op : == | !=
# role_op : == | != | eq | dom | domby | incomp
#
# names : name | { name_list }
```

[4] Linux UIDs and SELinux user identities ***should*** match because login and similar applications will try to look up the match. If it fails to find a match, it will fall back to user_u.

```
# name_list : name | name_list name#
#

#
# Restrict the ability to transition to other users
# or roles to a few privileged types.
#

constrain process transition
 ( u1 == u2 or t1 == privuser );

constrain process transition
 ( r1 == r2 or t1 == privrole );

#
# Restrict the ability to label objects with other
# user identities to a few privileged types.
#

constrain dir_file_class_set { create relabelto relabelfrom }
 ( u1 == u2 or t1 == privowner );

constrain socket_class_set { create relabelto relabelfrom }
 ( u1 == u2 or t1 == privowner );
```

## 2.12. Special Interfaces and File Systems

Some of these are discussed more extensively in other locations, and are here to highlight their nature. These are various special interfaces into the kernel and file system details.

**Tip**

> The shared library `libselinux` provides an abstraction layer for all of these interfaces. If you are writing an application, use this library instead of trying to directly access these interfaces. To see what is provided with `libselinux`, run the command `rpm -ql libselinux`. This will show all the utilities and associated manual pages included in the library.

- The special files at `/proc/<PID>/attr/` allow userspace access to context information about a process. `<PID>` is the process ID for the process you are examining. This access includes getting and setting security attributes for the process. These pseudo files expose the getting and setting:
  - `current` — current security context.
  - `prev` — the context prior to the last `exec`, which means the context of the process that called this process.
  - `exec` — the context to apply at the next `exec`
  - `fscreate` — the context to apply to any new files created by this process.

- The pseudo file system selinuxfs is mounted at `/selinux/`. It provides the SELinux policy API for userspace. Some of what `libselinux` abstracts from this pseudo file system is loading policy, enabling or disabling SELinux, and making AVC checks.

- Security file contexts are stored in the values in the *`security.selinux`* parameter of the file's extended attributes. This field is read when any subject makes a request for the SELinux type of a file. Extended attribute support is extremely limited for pseudo file systems at this time. Currently only devpts has support for xattrs, but work is ongoing to add further support for more pseudo file systems.

  As with the other special interfaces, it is recommended to use `libselinux` to interface with the functions.

# Chapter 3.

# Targeted Policy Overview

This chapter is an overview and examination of the targeted policy, which is the supported policy for Red Hat Enterprise Linux.

Much of the content in this chapter is applicable to all the kinds of SELinux policy, in terms of file locations and type of content in those files. What is different is which files exist in the key locations and what is in them.

As with Chapter 2 *SELinux Policy Overview*, you need to install both the policy source and binary packages for the targeted policy.

- `selinux-policy-targeted-sources-<`*`version`*`>`
- `selinux-policy-targeted-<`*`version`*`>`

**Important**

When you have the policy sources installed, `rpm` may assume that you have modified the policy and may not automatically load a newly installed policy. This occurs if you have ever loaded the policy from source, that is `make load`, `make reload`, or `make install`. New binary policy packages install `policy.<`*`version`*`>` as, for example, `$SELINUX_POLICY/policy.18.rpmnew`.

If you have not modified the policy or want to use the binary policy package, you can `mv policy.18.rpmnew policy.18`, then `touch /.autorelabel` and reboot. If you have modified the policy and want to load your modifications, you must upgrade the policy source package and `make load`. Policy building is discussed in Chapter 7 *Compiling SELinux Policy*.

If you have only built the policy but never loaded it, that is, only run `make policy`, you should not run into this situation. The binary policy package installs cleanly, having determined you are not running a custom policy.

Work is ongoing to improve package installation logic so the entire process is automated by `rpm`. Expect this to be included in a future update to Red Hat Enterprise Linux 4.

## 3.1. What is the Targeted Policy?

The SELinux policy is highly configurable. For Red Hat Enterprise Linux 4, Red Hat supports a single policy, the *targeted policy*. Under the targeted policy, every subject and object runs in the `unconfined_t` domain *except* for the specific targeted daemons. The objects on the system that are in the `unconfined_t` domain are allowed by SELinux to have no restrictions and fall back to using standard Linux security, that is, DAC. This policy is flexible enough to fit into enterprise infrastructures. The daemons that are part of the targeted policy run in their own domains and are restricted in every operation they perform on the system. This way daemons that are broken or exploited are limited in the damage they can do.

The opposite of the targeted policy is the *strict policy*. This does not ship with Red Hat Enterprise Linux. In the strict policy, every subject and object are in a specific security domain, with all interactions and transitions individually considered within the policy rules. This is a much more complex environment.

This guide focuses on the targeted policy that comes with Red Hat Enterprise Linux, and the components of SELinux used by the targeted daemons.

The targeted daemons are:

- dhcpd — this policy is dissected and explained in Chapter 4 *Example Policy Reference - dhcpd*.
- httpd
- mysqld
- named
- nscd
- ntpd
- portmap
- postgres
- snmpd
- squid
- syslogd
- winbind

The policy can be manipulated using command line or GUI tools. This is discussed extensively in Chapter 5 *Controlling and Maintaining SELinux*. Chapter 6 *Tools for Manipulating and Analyzing SELinux* and Chapter 7 *Compiling SELinux Policy* are two other chapters that detail working with the targeted policy.

## 3.2. Files and Directories of the Targeted Policy

These are common files and directories, and their purposes.

/etc/selinux/targeted/booleans

This is the default setting for the Booleans in the targeted policy:

```
cat /etc/selinux/targeted/booleans
allow_ypbind=1
dhcpd_disable_trans=1
httpd_disable_trans=0
httpd_enable_cgi=1
httpd_enable_homedirs=1
httpd_ssi_exec=1
httpd_tty_comm=0
httpd_unified=1
mysqld_disable_trans=0
named_disable_trans=0
named_write_master_zones=0
nscd_disable_trans=0
ntpd_disable_trans=0
portmap_disable_trans=0
postgresql_disable_trans=0
snmpd_disable_trans=0
squid_disable_trans=0
syslogd_disable_trans=0
winbind_disable_trans=0
ypbind_disable_trans=0
```

Using Boolean values to define the state of optional policy allows for the tunables to be switchable during runtime. The kernel accesses the state of the values in /selinux/booleans/*, with a separate file for each Boolean. If you run echo "1 1" > squid_disable_trans to turn off the targeted policy for squid by disabling the transition from unconfined_t to squid_t, you can then make the change take effect by running echo 1 > /selinux/commit_pending_bools. The value in

`/etc/selinux/targeted/booleans` would then change to `squid_disable_trans=1`. An easier technique for changing Booleans is to use the `setsebool` command.

If you change the value in `/etc/selinux/targeted/booleans`, the change takes effect upon next policy load, such as a reboot or `make load` (refer to Chapter 7 *Compiling SELinux Policy*).

Booleans work by having the `if` statements with conditional policy compiled into the binary policy, so the potential policy for each conditional is always present.

If you look at a pseudo file system Boolean file, for example `cat /selinux/booleans/httpd_unified/`, you get two values returned, 1 1. The first value represents the current value, the other is the pending value that is to be set programmatically when a `security_commit_booleans()` is run, that is, when policy is loaded. Another time this occurs is when you run `setsebool -P`. The `-P` writes all the pending Boolean values to the disk.

`/etc/selinux/targeted/contexts/`

This directory contains security context information used at run time by various applications, such as `restorecon`. Within `contexts/` are a number of files and directories. Here are the most important:

- `default_contexts` — this file defines the default security context(s) for local and remote user sessions, `cron` jobs, and so forth.
- `files/` — this subdirectory contains security context configuration files used by applications needing to set file labels during runtime, such as `rpm`, `restorecon`, `setfiles`, and `udev`.
- `userhelper_context` — this file sets the context for the `userhelper` application to use.

`$SELINUX_SRC/domains/program/`

The location of the TE files that define the policy for the daemons covered by the targeted policy. If a TE file is not in this directory, then it is not compiled into the policy.

`$SELINUX_SRC/file_contexts/`

All of the file contexts for the targeted and unconfined daemons are in the directory `file_contexts/program`. When the policy is built, all of the relevant `*.fc` files are concatenated into `$SELINUX_SRC/file_contexts/file_contexts`. A file contexts file is considered relevant to the policy if there is a corresponding `$SELINUX_SRC/domains/programs/*.fc` file. A copy of `file_contexts` is at `/etc/selinux/targeted/contexts/files/file_contexts`.

For files that are not part of the targeted daemons and their associated file contexts files, the file `types.fc` is referenced for setting the security context, especially for when the policy is installed or if the file system is relabeled.

This directory is discussed thoroughly in Section 3.3 *Understanding the File Contexts Files*.

`$SELINUX_SRC/file_contexts/distros.fc`

Each distribution of Linux that supports SELinux may have unique file contexts that should only be included if the policy is being compiled on that system. The set for Red Hat Enterprise Linux is grouped inside of `ifdef(‘distro_redhat’, ... ’)`‘, and includes contexts for Red Hat specific applications such as `system-config-securitylevel`, packages with possibly unique file locations, and file contexts for the `/emul` libraries for x86 emulation on 64-bit systems.

`$SELINUX_SRC/domains/unconfined.te`

This file defines the domain for unconfined processes, that is, everything that is not specifically a targeted daemon.

$SELINUX_SRC/appconfig/

This directory contains application configuration files that provide contexts or partial contexts for certain daemons and utilities. A partial context is when the user identity is not included. This identity is inferred from the user who runs the utility.

The kind of utilities that rely upon the appconfig contexts are crond, newrole, and login, which need to have a context that derives from a user rather than their own context. These files provide a list of possible contexts the program can try to set, and the policy decides if the process can transition to those contexts.

These various files are installed as the separate files and directories within $SELINUX_POLICY/contexts/, and are used in runtime by libselinux to search through for usable contexts.

In a stricter policy than the targeted policy, there would be additional entries since all users and daemons run in their own security context instead of unconfined_t. For example, when parsing through default_contexts, if the policy defines that a context is not allowed for a user, it would be ignored and the next one checked. This way the file can have a cascading set of partial contexts, so the most privileged gets the first choice, and the least privileged gets the last choice. In default_contexts for the targeted policy, the most and least privileged are the same

```
cat default_contexts
system_r:unconfined_t   system_r:unconfined_t
```

The default_type file is the configuration file for when applications need to know which domains are to be associated with which roles. In the targeted policy, there is effectively one single role for subjects: system_r. For example, newrole looks to this file to know what domains to assign each transitioned role:

```
cat default_type
system_r:unconfined_t
```

There ins only a partial context in failsafe_context. This is what is returned if default_contexts does not have an appropriate context. In other words, if nothing else matches, try this context. Note that it is the same context as in default_contexts. This file is more useful in a stricter policy.

```
cat failsafe_context
system_r:unconfined_t
```

When run_init executes a script in /etc/rc.d/, this is the context that run_init transitions to *before* running the script. This way, the context executing the scripts is the same as when they are executed by init.

```
cat initrc_context
user_u:system_r:unconfined_t
```

These are the default contexts applied to different media types, for example, when they are mounted on /media:

```
cat media
cdrom system_u:object_r:removable_device_t
floppy system_u:object_r:removable_device_t
disk system_u:object_r:fixed_disk_device_t
```

This context covers removable media types, such as USB flash storage devices:

```
cat removable_context
system_u:object_r:removable_t
```

The root_default_contexts allows login to root to be different than login to a normal user:

```
cat root_default_contexts
system_r:unconfined_t   system_r:unconfined_t
```

This is the context userhelper transitions to before executing the application that requires the privilege escalation:

```
cat userhelper_context
```

```
system_u:system_r:unconfined_t
```

`$SELINUX_SRC/types/*`

These files are the type declarations for general sets of types. The types are grouped by similarities such as being a file, being related to security, network, or devices. The name of the type declaration file reflects its contents.

One odd file included in the targeted policy is `$SELINUX_SRC/types/apache.te`. The file contains this one line macro:
`define('admin_tty_type', '{ tty_device_t devpts_t }')`

This macro is connected with a conditional set of rules in the `httpd` TE file at `$SELINUX_SRC/domains/program/apache.te`. The confitional rules allow `httpd` to utilize the console (`if (httpd_tty_comm) {}`). This allows Apache HTTP to use the console for parts of the SSL certification handling process.

The reason the macro defining `admin_tty_type` is in `types/apache.te` is that the macro is included in the targeted policy only for the benefit of `httpd`. Apache HTTP needs this macro defined for the `httpd` policy to work.

In a stricter policy, the system administrator domain `sysadm_t` is used, and it's associated TE file at `/etc/selinux/strict/src/policy/domains/admin.te` supplies the `admin_tty_type` macro.

The file `$SELINUX_SRC/types/files.fc` defines the contexts for all of the file types on the system.

`$SELINUX_SRC/domains/program/*`

These are the TE policy files that make the targeted daemons protected. In SELinux, in the tree at `$SELINUX_SRC/domains/` are all the rules that govern the behavior of the various domains. If a particular `*.te` is not in the `$SELINUX_SRC/domains/` path, it is not compiled in as part of the policy.

In Chapter 4 *Example Policy Reference - `dhcpd`*, the policy for `dhcpd` is completely dissected and examined as a reference for all of the policy files for the targeted daemons.

`$SELINUX_SRC/assert.te`, `$SELINUX_SRC/attrib.te`, and `$SELINUX_SRC/constraints`

The file `assert.te` contains the `neverallow` assertions, discussed in Section 2.8 *TE Rules - Access Vectors*. The attributes declared for the targeted policy are in `attrib.te`, discussed in Section 2.6 *TE Rules - Attributes*. Constraining rules, as discussed in Section 2.11 *TE Rules - Constraints*, are defined for the targeted policy in the file `constraints`.

`$SELINUX_SRC/flask/`

This directory is where several important definitions occur. In `access_vectors`, object classes are defined, as discussed in Section 2.5 *Object Classes and Permissions*. The file `initial_sids` provides the booting kernel with the initial security identifiers to use until policy can be loaded, as described in Section 2.3 *Policy Role in Boot*. Security object classes are defined in `security_classes`. The shell scripts and `Makefile` are used in SELinux kernel development, and are not intended for end-user usage.

`$SELINUX_SRC/macros/`

Macros are discussed in Section 2.9 *Policy Macros*. Only two macro files in this directory are used, `core_macros.te` and `global_macros.te`. The directory `$SELINUX_SRC/macros/program/` contains the macro files for various daemons. Only the macro files that correspond to a `*.te` file in `$SELINUX_SRC/domains/program/` are actually used in the policy.

`$SELINUX_SRC/genfs_contexts`

As explained in Section 2.4 *File System Security Contexts*, this file supplies the contexts for mountpoint labeling, where a mounted file system is given a single, overarching context instead of an individual context for each file.

`$SELINUX_SRC/initial_sid_contexts`

These are the security contexts that are applied to the initial contexts in `$SELINUX_SRC/flask/initial_sids` and are used by the kernel during boot before it has loaded the policy. Refer to Section 2.3 *Policy Role in Boot* for more information.

`$SELINUX_SRC/mls`

This file is unused in the targeted policy, but is noteworthy for those interested in MLS security. Refer to Chapter 9 *References* for sources of information about MLS.

`$SELINUX_SRC/net_contexts`

This file has the contexts for network entities, with many declarations within an `ifdef` statement that depends on the presence of a specific `*.te` file in `$SELINUX_SRC/domains/program/`. The syntax looks like this:

```
portcon <protocol> <{ port | port-range }> <type>
```

When invoked, a network context declaration looks like this:

```
ifdef(‘mta.te’, ‘
portcon tcp 25 system_u:object_r:smtp_port_t
portcon tcp 465 system_u:object_r:smtp_port_t
portcon tcp 587 system_u:object_r:smtp_port_t
’)
...
ifdef(‘use_dhcpd’, ‘portcon udp 67  \
  system_u:object_r:dhcpd_port_t’)
...
# Defaults for reserved ports.  Earlier portcon entries take
# precedence; these entries just cover any remaining reserved
# ports not otherwise declared or omitted due to removal of a
# domain.
portcon tcp 1-1023 system_u:object_r:reserved_port_t
portcon udp 1-1023 system_u:object_r:reserved_port_t
...
netifcon eth0 system_u:object_r:netif_eth0_t \
  system_u:object_r:unlabeled_t
```

`$SELINUX_SRC/policy.conf`

This file is created by `m4` during the policy compiling process. It is all of the TE rules from `domains/` with the macros expanded, and the result concatenated together. The compilation process is covered in Chapter 7 *Compiling SELinux Policy*, and you can learn about analyzing the policy using `policy.conf` in Chapter 6 *Tools for Manipulating and Analyzing SELinux*.

`$SELINUX_SRC/rbac`

This file defines which roles are allowed to attain which other roles. Roles are discussed in Section 2.10 *SELinux Users and Roles*. These are all the allowed role transitions in the targeted policy: This file only specifies which roles may transition to which other roles, it does not grant permission to actually change role.

```
allow sysadm_r system_r;
allow user_r system_r;
allow user_r sysadm_r;
allow sysadm_r user_r;
allow system_r sysadm_r;
```

`$SELINUX_SRC/tunables/`

The *tunable* is a way of switching on or off certain settings that have global effect. For example, the file `distro.tun` has only one Linux distribution defined, the others are `dnl define`:

```
define('distro_redhat')
```

The existence of this definition triggers conditional statements in the TE files for `httpd`, `mysqld`, `named`, and `snmpd` in `$SELINUX_SRC/domains/program`, as well as `$SELINUX_SRC/macros/program/userhelper_macros.te`.

Tunables are included in the policy at compile time and are not a flexible way to manage settings that you want to effect more immediately. For the most part, the tunables have been replaced by Booleans in `/etc/selinux/targeted/booleans` that are checked during runtime.

The second file, `tunable.tun`, has several definitions which are in use in the targeted policy:

```
define('targeted_policy')
define('nscd_all_connect')
define('nfs_home_dirs')
```

The `targeted_policy` tunable is used by `apache.te`, `named.te`, `squid.te`, and `mta.te` in `$SELINUX_SRC/domains/programs/`, as well as `global_macros.te` and `apache_macros.te`. For example, this statement from `apache.te` is triggered to be included in the policy if `targeted_policy` is defined:

```
ifdef('targeted_policy', '
typealias httpd_sys_content_t alias httpd_user_content_t;
typealias httpd_sys_script_exec_t alias \
  httpd_user_script_exec_t;

if (httpd_enable_homedirs) {
allow httpd_sys_script_t user_home_dir_t:dir { getattr \
  search };
allow httpd_t user_home_dir_t:dir { getattr search };
}
') dnl targeted policy
```

The type aliases created support for Apache HTTP CGI scripting by users, aliasing the user equivalent of the `httpd` scripting type. Notice the `if (httpd_enable_homedirs)` statement. This is the Boolean value `httpd_enable_homedirs`, used for enabling public HTML directories being served from user home directories.

`$SELINUX_SRC/users`

This file contains the definitions for the SELinux users, as explained in Section 2.10 *SELinux Users and Roles* and Section 3.5 *Understanding the Roles and Users in the Targeted Policy*.

If you are trying to run a minimal policy to reduce disk and memory usage, you can try removing unused files from `$SELINUX_SRC/domains/program/`. A TE file may be unused if the daemon associated with that domain file is not installed. For example, if you do not have the nameserver BIND installed, you may be able to remove the associated policy by moving the file `$SELINUX_SRC/domains/program/named.te`. This reduces the SELinux footprint in kernel memory and possibly some impact on performance.

After you remove the `*.te` file from the directory, you need to `cd $SELINUX_SRC/` and `make load`. This takes effect immediately. Policy compiling is discussed in detail in Chapter 7 *Compiling SELinux Policy*. If you move the file to `$SELINUX_SRC/domains/program/unused/`, the TE policy is easy to obtain should you choose to install BIND at a later date.

**Warning**

> Removing the wrong file can result in your system being unable to boot in enforcing mode. Policy compilation can fail if dependencies are not available. Be sure you know the consequences of removing any of the `*.te` files from `/etc/selinux/targeted/src/policy/`.
>
> A better solution for most cases is to use the Booleans to disable the policy for uninstalled applications. This compromise reduces some of the kernel overhead

Here is an abbreviated file tree for the policy source. Not included are the TE files that are unused in the targeted policy. Note the presence of the files `policy.conf`, `file_contexts/file_contexts`, and `tmp/*`. These indicate a policy that has been compiled from source and possibly loaded.

```
tree /etc/selinux/targeted/src/policy/
/etc/selinux/targeted/src/policy/
|-- COPYING
|-- ChangeLog
|-- Makefile
|-- README
|-- VERSION
|-- appconfig
|   |-- default_contexts
|   |-- default_type
|   |-- failsafe_context
|   |-- initrc_context
|   |-- media
|   |-- removable_context
|   |-- root_default_contexts
|   ‘-- userhelper_context
|-- assert.te
|-- attrib.te
|-- constraints
|-- domains
|   |-- misc
|   |   ‘-- unused
|   |-- program
|   |   |-- apache.te
|   |   |-- dhcpd.te
|   |   |-- hotplug.te
|   |   |-- init.te
|   |   |-- initrc.te
|   |   |-- ldconfig.te
|   |   |-- mailman.te
|   |   |-- modutil.te
|   |   |-- mta.te
|   |   |-- mysqld.te
|   |   |-- named.te
|   |   |-- nscd.te
|   |   |-- ntpd.te
|   |   |-- portmap.te
|   |   |-- postgresql.te
|   |   |-- rpm.te
|   |   |-- snmpd.te
|   |   |-- squid.te
|   |   |-- syslogd.te
|   |   |-- udev.te
|   |   |-- winbind.te
|   |   ‘-- ypbind.te
|   ‘-- unconfined.te
|-- file_contexts
|   |-- distros.fc
|   |-- file_contexts
|   |-- misc
|   |-- program
|   |   |-- apache.fc
|   |   |-- dhcpd.fc
|   |   |-- hotplug.fc
|   |   |-- init.fc
|   |   |-- initrc.fc
|   |   |-- ldconfig.te
|   |   |-- mailman.fc
|   |   |-- modutil.fc
|   |   |-- mta.fc
|   |   |-- mysqld.fc
|   |   |-- named.fc
|   |   |-- nscd.fc
|   |   |-- ntpd.fc
|   |   |-- portmap.fc
|   |   |-- postgresql.fc
|   |   |-- rpm.fc
|   |   |-- snmpd.fc
|   |   |-- squid.fc
|   |   |-- syslogd.fc
|   |   |-- udev.fc
|   |   |-- winbind.fc
|   |   |-- ypbind.fc
|   ‘-- types.fc
|-- flask
|   |-- Makefile
|   |-- access_vectors
|   |-- initial_sids
|   |-- mkaccess_vector.sh
|   |-- mkflask.sh
|   ‘-- security_classes
|-- fs_use
|-- genfs_contexts
|-- initial_sid_contexts
|-- macros
|   |-- core_macros.te
|   |-- global_macros.te
|   ‘-- program
|       |-- apache_macros.te
|       |-- mta_macros.te
|       |-- sendmail_macros.te
|       ‘-- ypbind_macros.te
|-- mls
|-- net_contexts
|-- policy.conf
|-- rbac
|-- serviceusers
|-- tmp
|   |-- load
|   ‘-- program_used_flags.te
|-- tunables
|   |-- distro.tun
|   ‘-- tunable.tun
|-- types
|   |-- apache.te
|   |-- device.te
|   |-- devpts.te
|   |-- file.te
|   |-- network.te
|   |-- nfs.te
|   |-- procfs.te
|   |-- security.te
|   '-- x.te
'-- users
```

## 3.3. Understanding the File Contexts Files

The files in `$SELINUX_SRC/file_contexts/` declare the security contexts that are applied to files when the policy is installed. You can read more about what a file context is at Section 2.4 *File System Security Contexts*.

The file context descriptions use regular expression pattern matching (*regexp*) to match a file, set of files, directory, or directory and associated files. A specific SELinux label is then applied to that:

```
# Syntax of file context description

regexp <-type> ( <file_label> | <<none>> )
```

The regexp is anchored on both ends, meaning the expression search only considers matches that start with the first character and end with the last character. This means that it ignores an expression that appears in the middle of a sentence, the way `/var/run` appears in this sentence, and only declares a match if the pattern is on a line by itself:

```
/var/run
```

This is the way a directory is displayed by the output of `ls`, and is how `setfiles` sees files and directories when it traverses the directory tree. In regexp notation, this kind of match is denoted by prepending a `^` (*caret*) symbol and appending a `$` (dollar sign or *ding*) to the expression. This is done automatically by SELinux, and can be overridden using the match-anything pattern `.*` on either or both sides of the regexp pattern.

The field `-type` is optional and can be left blank. When it is filled, it is similar to the mode field for the `ls` command. For example, the `-d` means to match only directories, the `--` means to match only files.

The value field is the last field on the right, and is set to either a single security label such as `system_u:object_r:home_root_t` or is set to <<none>>. The <<none>> value tells the relabeling application to not relabel the matching file. In the case where there is more than one match, the last matching value is used. Hard linked files that have different contexts generate a labeling error, and the file is labeled based on the last matching specification other than <<none>>.

There are files listed in `types.fc` that are not persistent, but get created each time during boot. Those files gain their labels through type transition rules, but they are listed here to prevent their label being overwritten by a relabeling operation during runtime. One example of this is `/var/run/utmp`.

Here are some examples from `$SELINUX_SRC/file_contexts/types.fc`, the main file describing file contexts:

```
/bin(/.*)?   system_u:object_r:bin_t
/bin/bash  -- system_u:object_r:shell_exec_t

/u?dev(/.*)?   system_u:object_r:device_t
/u?dev/pts(/.*)?  <<none>>
ifdef('distro_redhat', '
/dev/root  -b system_u:object_r:fixed_disk_device_t
')

/proc(/.*)?   <<none>>
/sys(/.*)?   <<none>>
```

```
/selinux(/.*)?   <<none>>
/etc(/.*)?   system_u:object_r:etc_t
/etc/passwd\.lock -- system_u:object_r:shadow_t
/etc/group\.lock -- system_u:object_r:shadow_t
/etc/shadow.*  -- system_u:object_r:shadow_t

/usr(/.*)?/lib(64)?/.*\.so(\.[^/]*)* -- system_u:object_r:shlib_t
/usr(/.*)?/java/.*\.so(\.[^/]*)* -- system_u:object_r:shlib_t
```

Similarly, there are specific file contexts files for all domains, depending on their special needs. The `*.fc` files are present in the policy source, but are only used if there is an associated `*.te` file in `$SELINUX_SRC/domains/program/`. Here is an example from `mta.fc`:

```
# types for general mail servers
/usr/sbin/sendmail(.sendmail)? -- system_u:object_r:sendmail_exec_t
/usr/lib(64)?/sendmail         -- system_u:object_r:sendmail_exec_t
/etc/aliases            --      system_u:object_r:etc_aliases_t
/etc/aliases\.db        --      system_u:object_r:etc_aliases_t
/var/spool/mail(/.*)?           system_u:object_r:mail_spool_t
/var/mail(/.*)?                 system_u:object_r:mail_spool_t
```

```
ifdef(‘dhcp_defined’, ‘’, ‘
/var/lib/dhcp(3)? -d system_u:object_r:dhcp_state_t
define(‘dhcp_defined’)
’)
```

**Example 3-1. `ifdef` statement in a context file**

Another interesting example is Example 3-1. The file context `dhcp_state_t` must be set for the DHCP-based daemons to work properly, so the pattern and value are in both `dhcpd.fc` and `dhcpc.fc` (the DHCP client daemon contexts). This is to ensure the label value is present in case one file is not included in a policy build. In fact, this is the case for the targeted policy, where the DHCP client is not confined by SELinux policy. In this case, the file `dhcpd.fc` declares the file context for `/var/lib/dhcp` for the pattern matching `^/var/lib/dhcp(3)?$`.

If you include policy to cover the DHCP client programs, you want to ensure that it also can declare the context for `/var/lib/dhcp/`. However, if you include both programs and they both declare the context, `setfiles` reports an error during policy build. This happens when a file context is specified multiple times, even if the specification is identical.

To handle this, a conditional `ifdef` statement is used. When concatenating the file context files into the single file `$SELINUX_SRC/file_contexts/file_contexts`, the first time the `ifdef` statement is reached, the definition `dhcp_defined` is checked for. If it is defined, the value true is returned, and the additional file context is skipped. If `dhcp_defined` has not been defined, the value returns as false, the file context is read from inside the statement, and `dhcp_defined` is defined.

## 3.4. Common Macros in the Targeted Policy

Macros in SELinux are discussed in Section 2.9 *Policy Macros*. This section covers macros that are used extensively throughout the targeted policy. These were chosen by frequency, the list comprising mainly macros used nine or more times in the various policy files. A large number of macro files are present in the policy, but are not necessarily called by any of the TE files. There are macro files for many, but not all, of the targeted daemons.

daemon_domain, daemon_base_domain, and daemon_core_rules

The macro daemon_domain is in $SELINUX_SRC/macros/global_macros.te, and is common to all of the targeted daemons. The purpose of daemon_domain is to group together permission needs common to all daemons. These needs include creating a process ID (PID) file and running df to check disk usage. In addition, two macros are called, daemon_base_domain and read_locale.

The base common set of type declarations and permissions is defined in daemon_base_domain, and include allowing you to define a tunable that can disable the domain transition. You evoke one of these tunables when you set the Boolean value to disable the transition to one of the targeted domains, removing SELinux protection from that single daemon. Finally, daemon_core_rules is called.

This central macro is where the daemon's top-level domains and roles are declared:

```
define(‘daemon_core_rules’, ‘
type $1_t, domain, privlog $2;
type $1_exec_t, file_type, sysadmfile, exec_type;
```

daemon_core_rules gives a daemon the right to inherit and use descriptors from init, calls the uses_shlib() macro for the domain to use shared libraries, allows for common self signaling, and so forth.

can_network

Providing a top-level entry point for common networking policy, this macro appears in $SELINUX_SRC/macros/global_macros.te. One primary allow rule gives the domain access to TCP and UDP sockets to create, send and receive on a network interface from any node on any port. Read permission is granted for network files, which are configuration files in /etc/ that network daemons need, mainly /etc/resolv.conf:

```
# can_network(domain)
define(‘can_network’,‘
allow $1 self:udp_socket create_socket_perms;
allow $1 self:tcp_socket create_stream_socket_perms;
allow $1 netif_type:netif { tcp_send udp_send rawip_send };
allow $1 netif_type:netif { tcp_recv udp_recv rawip_recv };
allow $1 node_type:node { tcp_send udp_send rawip_send };
allow $1 node_type:node { tcp_recv udp_recv rawip_recv };
allow $1 port_type:{ tcp_socket udp_socket } { send_msg \
  recv_msg };
...
allow $1 net_conf_t:file r_file_perms;
’)dnl end can_network definition
```

The limitations on which nodes and ports are allowed by domain are defined in separate rules. Recall that by default, everything is denied in SELinux. The allow rules here grant only the permission to, for example, make a bind(2) call to the socket, but the specific port binding requires another authorization. The permission name_bind to bind to the port is still limited by the domain, so that, for example, named may be allowed by standard Linux permissions to bind to port 22, but SELinux blocks access and generates an avc: denied message in $AUDIT_LOG. This is because named_t is not allowed to bind to a port of type ssh_port_t, which is the type for the SSH port.

can_unix_connect

This popular macro from core_macros.te provides permissions for establishing a UNIX stream connection:

```
# can_unix_connect(client, server)
define(‘can_unix_send’,‘
allow $1 $2:unix_dgram_socket sendto;
’)
```

The can_unix_connect macro handles the control between the two domains. In addition, the socket needs write permission to the file associated with the socket. For this reason the can_unix_connect macro is paired with other allow rules in the policy:

```
# From $SELINUX_SRC/domains/program/syslogd.te

allow privlog devlog_t:sock_file rw_file_perms;
...
can_unix_connect(privlog, syslogd_t)
```

create_dir_file

Common permissions for creating directories, regular files, and symlinks are grouped into this macro from core_macros.te:

```
define(‘create_dir_file’, ‘
allow $1 $2:dir create_dir_perms;
allow $1 $2:file create_file_perms;
allow $1 $2:lnk_file create_lnk_perms;
’)
```

create_socket_perms

This is a single line macro from core_macros.te that expands into a list of permissions needed to create and manage sockets. This macro is inserted directly into a rule, and gives a single location to define this common permissions set. This is the macro followed by an example rule evoking the macro:

```
define(‘create_socket_perms’, ‘{ create ioctl read getattr \
  write setattr append bind connect getopt setopt shutdown }’)

allow winbind_t self:unix_dgram_socket create_socket_perms;
```

domain_auto_trans and domain_trans

When you want a particular transition to be the default transition behavior for a domain, use domain_auto_trans, from core_macros.te. It calls domain_trans() to do the actual work. If you just want to allow a transition to take place and handle the context with setexeccon(), you can use just a domain_trans():

```
# domain_auto_trans(parent_domain, program_type, child_domain)
define(‘domain_auto_trans’,‘
domain_trans($1,$2,$3)
type_transition $1 $2:process $3;
’)
```

The domain_trans macro allows the parent process a typical set of permissions to transition to the new domain. It sets a few dontaudit rules, allows the parent process to execute the program, allows the child to reap the new domain and exchange and use file descriptions with the parent process, as well as write back to the old domain via a named pipe (FIFO). Finally, the new domain is given permission to read and execute the program, making the program the entry point for the domain.

Because only one transition can be the default for a given pair of types, you can have one domain_auto_trans rule followed by multiple domain_trans rules that allow other options. These can be used by a security-aware application.

file_type_auto_trans and file_type_trans

From core_macros.te, file_type_trans allows the permissions for the given domain to create files in the given parent directory that have the given file type. These givens are inserted as the variables $1 and so forth. To make a particular transition the default behavior, use file_type_auto_trans().

This macro has a built-in conditional that it can take three or four parameters in defining the output type_transition rule:

```
# file_type_auto_trans(creator_domain, parent_directory_type, \
  file_type, object_class)
#
# the object class will default to notdevfile_class_set if not
# specified as the fourth parameter
define(‘file_type_auto_trans’,‘
ifelse(‘$4’, ‘’, ‘
file_type_trans($1,$2,$3)
type_transition $1 $2:dir $3;
type_transition $1 $2:notdevfile_class_set $3;
’, ‘
file_type_trans($1,$2,$3,$4)
type_transition $1 $2:$4 $3;
’)dnl end ifelse

’)
```

The `file_type_trans` allows the process to modify the directory and create the file, all with the proper labeling. As with `domain_auto_trans`, you can specify additional allowed options for use by security-aware applications that can call `setexeccon()`.

The optional fourth parameter, `$4`, lets you specify a particular file object class. The default is non-device files, `notdevfile_class_set()`.

`r_dir_perms`, `r_file_perms`, `rw_file_perms`, and `ra_file_perms`

These single line macros from `core_macros.te` are used directly in TE rules to group common permission sets depending on the need to read, write, append, and execute files and directories:

```
# Permissions for reading directories and their attributes.
#
define(‘r_dir_perms’, ‘{ read getattr lock search ioctl }’)

#
# Permissions for reading files and their attributes.
#
define(‘r_file_perms’, ‘{ read getattr lock ioctl }’)

#
# Permissions for reading and writing files and their
# attributes.
#
define(‘rw_file_perms’, ‘{ ioctl read getattr lock write \
  append }’)

#
# Permissions for reading and appending to files.
#
define(‘ra_file_perms’, ‘{ ioctl read getattr lock append }’)
```

`tmp_domain`

This macro identifies the domain as needing to be able to create temporary files in `/tmp/`. A file transition is setup for temporary files created by the domain. A separate temporary type, `$1_tmp_t`, is declared. This type is used to label temporary files created by the domain so that each domain may only read and write its own temporary files. Additional rules may be written that allow permissions for domains to control temporary files of other domains, such as allowing `tmpwatch` to clean up `/tmp/`. Most of the targeted daemons use this macro:

```
define(‘tmp_domain’, ‘
type $1_tmp_t, file_type, sysadmfile, tmpfile $2;
file_type_auto_trans($1_t, tmp_t, $1_tmp_t)
’)
```

## 3.5. Understanding the Roles and Users in the Targeted Policy

Building on your understanding from reading Section 2.10 *SELinux Users and Roles*, this section covers the specific roles enabled for the targeted policy. As explained previously, roles are not very active in the targeted policy, but can be an essential part of a localized SELinux installation. You can see that `unconfined_t` is in every role, which significantly reduces the usefulness of roles in the targeted policy. Using roles more extensively requires a change to the strict policy paradigm, where every process runs in an individually considered domain.

Effectively, there are only two roles in the targeted policy: `system_r` and `object_r`. The initial role is `system_r`, and everything else inherits that role. The remaining roles are defined for compatibility purposes between the targeted policy and strict policy.[1]

Of the four roles, three are defined by the policy. The role `object_r` is an implied role and is not found in policy source. Since roles are created and populated by types through one or more declarations in `$SELINUX_SRC/domains/*`, there is not a single file that declares all the roles. To generate the lists of types and their roles in this section, the policy analysis tool **apol** was used. Usage of this tool is explained in Section 6.3 *Using **apol** for Policy Analysis*.

`system_r`

This role is for all system processes except user processes:

```
unconfined_t
httpd_t
httpd_sys_script_t
httpd_suexec_t
httpd_php_t
httpd_helper_t
dhcpd_t
ldconfig_t
mailman_queue_t
mailman_mail_t
mailman_cgi_t
system_mail_t
mysqld_t
named_t
ndc_t
nscd_t
ntpd_t
portmap_t
postgresql_t
snmpd_t
squid_t
syslogd_t
winbind_t
ypbind_t
```

`user_r`

This is the default user role for regular Linux users. In a strict policy, individual users might be used, allowing for the users to have special roles to do privileged operations. In the targeted policy, all users run in a single domain:

```
unconfined_t
```

[1] Any of the roles could have been chosen for the targeted policy, but `system_r` already had existing authorization for the daemon domains, simplifying the process. This was done in this fashion because there is no role aliasing mechanism.

`object_r`

> In SELinux, roles are not utilized for objects when RBAC is being used. Roles are strictly for subjects. This is because roles are task-oriented and they group together doers, which are subjects. For this reason, all objects universally have the role `object_r`, and the role is only used as a placeholder in the label.

`sysadm_r`

> This is the system administrator role in a strict policy. In such a policy, switching to the root user with `su` gives you the role of `sysadm_r`. However, if you login directly as the root user, you may default to `staff_r` and still need to run `newrole -r sysadm_r` before doing many system administration tasks. In the targeted policy, the following retain `sysadm_r` for compatibility:

```
unconfined_t
httpd_sys_script_t
httpd_helper_t
ldconfig_t
ndc_t
```

Similar to the situation for roles, there is effectively only one user identity in the targeted policy. The identity `user_u` was chosen because `libselinux` falls back to `user_u` as the default SELinux user identity. This occurs when there is no matching SELinux user for the Linux user who is logging in. Using `user_u` as the single user in the targeted policy makes it easier to switch to the strict policy. The remaining users exist for compatibility with the strict policy.[2]

The one exception is the SELinux user `root`. This user identity is picked up by login programs such as `su`, and you may notice `root` as the user identity in a process's context. This occurs when the SELinux user `root` starts daemons from the command line, or restarts a daemon originally started by `init`.

Here is a brief look at the `$SELINUX_SRC/users` file. Comments in `/* */` are annotations from this guide, all other content is original source from the `users` file.

```
src/policy/users
# Each user has a set of roles that may be entered by
# processes with the users identity.  The syntax of a user
# declaration is: #

#       user username roles role_set [ ranges MLS_range_set ];

# system_u is the user identity for system processes and
# objects.  There should be no corresponding Unix user
# identity for system_u, and a user process should never be
# assigned the system_u user identity.

  user system_u roles system_r;
/*     ^- user name      ^- the role user name can have     */

# user_u is a generic user identity for Linux users who have
# no SELinux user identity defined.  Authorized for all
# roles in the (targeted) policy.  sysadm_r is retained for
# compatibility, but could be dropped as long as userspace
# has no hardcoded dependency on it.  user_u must be
# retained due to present userspace hardcoded dependency.

  user user_u roles { user_r sysadm_r system_r };
/*     ^-user name  ^-set of roles the user name can have  */
```

[2] A user aliasing mechanism would work here, as well, to alias all identities from the strict policy to a single user identity in the targeted policy.

```
# root is retained as a separate user identity simply as a
# compatibility measure with the "strict" policy.  It could
# be dropped and mapped to user_u but this allows existing
# file contexts that have "root" as the user identity to
# remain valid.

  user root roles { user_r sysadm_r system_r };
/*     ^user name ^- set of roles                          */
```

# Chapter 4.

# Example Policy Reference - dhcpd

This chapter provides an understanding of how the policy works with the dhcpd daemon. This daemon ships as part of the dhcp package. This chapter first discusses the locations and purposes of key policy files, and then policy types are explained. This chapter serves as a reference analysis that can be applied to all of the targeted daemons. Analysis in this file results from direct investigation of the policy files as well as extensive usage of **apol**, which is discussed in Chapter 6 *Tools for Manipulating and Analyzing SELinux*.

## 4.1. Policy File Locations

This section covers the various top level files that comprise the policy for dhcpd. Refer to Section 4.2 *Policy Types - dhcpd* for a description of what the types are allowed to do.

$SELINUX_POLICY/domains/program/dhcpd.te

> This file defines the policy rules for the dhcpd domain, dhcpd_t. These rules are discussed in Section 4.2 *Policy Types - dhcpd*. Because the type enforcement file calls macros that are defined elsewhere, the dhcpd.te file is only the starting point for the policy. The policy building process expands the macros into many more lines of rules.

$SELINUX_POLICY/file_contexts/program/dhcpd.fc

> This defines the security context for files associated with the dhcpd server daemon, assigning them one of the dhcp_<*>_t types:

```
# dhcpd
/etc/dhcpd.conf  -- system_u:object_r:dhcp_etc_t
/etc/dhcp3(/.*)?  system_u:object_r:dhcp_etc_t
/usr/sbin/dhcpd.* -- system_u:object_r:dhcpd_exec_t
/var/lib/dhcp(3)?/dhcpd\.leases.* -- \
  system_u:object_r:dhcpd_state_t
/var/run/dhcpd\.pid -d system_u:object_r:dhcpd_var_run_t
ifdef(‘dhcp_defined’, ‘’, ‘
/var/lib/dhcp(3)? -d system_u:object_r:dhcp_state_t
define(‘dhcp_defined’)
’)
```

> **Note**
>
> As you are looking for dhcpd.fc, you see there are a large number of file contexts files in $SELINUX_POLICY/file_contexts/program/. Most of these files are unused. The context files are not pulled into the policy without a corresponding TE file in the $SELINUX_POLICY/domains/ path.

The context file contains an ifdef statement; the purpose here is to make certain the shared directory /var/lib/dhcp is available without declaring it multiple times. This is discussed in detail in Example 3-1.

## 4.2. Policy Types - dhcpd

This section discusses the types associated with the dhcpd policy.

**Note**

SELinux policy uses a number of macros written in the m4 macro language to make policy writing easier. In a type enforcement file such as dhcpd.te, macros are used extensively to call common capabilities for subjects and targets. These are discussed in Section 2.9 *Policy Macros* and Section 3.4 *Common Macros in the Targeted Policy*.

For the purposes of dissecting the dhcpd policy, this section is based on what is found in the policy.conf file. Since this file is created by the build process, the macros have been expanded entirely. It takes some practice, but soon you can find and understand the macros and the associated rulesets in the TE files from $SELINUX_SRC/domain/programs/.

dhcpd_t

This is the main, top-level domain for the dhcpd daemon. Nearly every rule in $SELINUX_SRC/domain/programs/dhcpd.te deals with this type, most notably the macros that expand into numerous rules. A complete list can be obtained using the **apol** tool. This is discussed further in Chapter 6 *Tools for Manipulating and Analyzing SELinux*. Some highlighted rules are:

- Various specific manipulations of the dhcpd_*_t domains, as explained below in further examples under each context..
- Network rules necessary for dhcpd to do its work, such as tcp_recv, udp_recv, and rawip_recv to network interfaces. Some examples are:

```
allow dhcpd_t netif_type : netif { tcp_send udp_send
  rawip_send };
allow dhcpd_t node_type : node { tcp_recv udp_recv \
  rawip_recv };
```

- Socket rules needed by dhcpd to create, listen, connect, accept, bind, read, write, control input and output (ioctl), get (getattr) and set (setattr) attributes, send (send_msg) and receive (recv_msg) messages, get (getopt) and set (setopt) command options, and so forth. Socket objects controlled are tcp_socket, udp_socket, netlink_route_socket, rawip_socket, unix_dgram_socket, unix_stream_socket, and reserved_port_socket. These are all object classes that SELinux controls the access to. Some examples are:

```
allow dhcpd_t node_type : { tcp_socket udp_socket } \
  node_bind ;
allow dhcpd_t port_type : { tcp_socket udp_socket } \
  { send_msg recv_msg };
allow dhcpd_t port_type : { tcp_socket udp_socket } \
  { send_msg recv_msg };
allow dhcpd_t self : rawip_socket { create ioctl read \
  getattr write setattr append bind connect getopt \
  setopt shutdown };
allow dhcpd_t self : tcp_socket { create ioctl read \
  getattr write setattr append bind connect getopt \
  setopt shutdown listen accept };
allow dhcpd_t self : udp_socket { create ioctl read \
  getattr write setattr append bind connect getopt \
  setopt shutdown };
allow dhcpd_t self : unix_dgram_socket { create \
  ioctl read getattr write setattr append bind \
  connect getopt setopt shutdown };
allow dhcpd_t self : unix_stream_socket { create \
  ioctl read getattr write setattr append bind \
  connect getopt setopt shutdown };
```

- As a network service, dhcpd is allowed to open a TCP or UDP socket to send and receive messages from any port. The attribute port_type covers a long list of ports: dns_port_t, dhcpd_port_t, http_cache_port_t, port_t, reserved_port_t, http_port_t, pxe_port_t, smtp_port_t, mysqld_port_t, rndc_port_t, ntp_port_t, portmap_port_t, postgresql_port_t, snmp_port_t, syslogd_port_t. The rule looks like this:

  ```
  allow dhcpd_t port_type:{ tcp_socket udp_socket } \
    { send_msg recv_msg };
  ```

- This rule allows dhcpd to control the dhcpd_t type, that is, itself, for process signaling:

  ```
  allow dhcpd_t self : process { sigchld sigkill sigstop \
    signull signal fork };
  ```

dhcpd_exec_t

This is the file type for the dhcpd executable. This type is the entry point for the dhcpd_t domain.

dhcpd_port_t

The dhcpd_port_t type has one direct rule governing it:

```
allow dhcpd_t dhcpd_port_t : udp_socket name_bind;
```

The daemon with the domain of dhcpd_t, that is, dhcpd, has the permission to bind to the object class of udp_socket, which opens a UDP port. Policy states that UDP port 67 is created with the domain of dhcpd_port_t:

```
grep dhcpd_port_t $SELINUX_SRC/net_contexts
ifdef(‘use_dhcpd’, ‘portcon udp 67 system_u:object_r:\
  dhcpd_port_t’)
```

SELinux has controls for port binding, meaning it is able to allow or deny port binding requests based on security labels. However, SELinux only controls attempts to bind to reserved ports, which are ports less than 1024, and to ports outside of the local port range, which is set in /proc/sys/net/ipv4/ip_local_port_range.

If dhcpd tries to bind to any port other than 67 port that is reserved or outside of the local range, the daemon is denied. This is because dhcpd_t is only allowed to bind to a port with the type of dhcpd_port_t, and only one port has that type, port 67.

dhcpd_state_t

This type dhcpd_state_t is the file type for the dhcpd lease file located at /var/lib/dhcp/dhcpd.leases. The dhcpd daemon is allowed to create, read, write, etc. a file with the context of dhcpd_state_t:

```
allow dhcpd_t dhcpd_state_t : file { create ioctl read \
  getattr lock write setattr append link unlink rename };
type_transition dhcpd_t dhcp_state_t : file dhcpd_state_t;
```

The second rule is a type_transition rule that comes from a macro defined in $SELINUX_SRC/macros/core_macros.te and used in the dhcpd.te file, file_type_auto_trans(dhcpd_t, dhcp_state_t, dhcpd_state_t, file).

This rule ensures that unless explicitly overwritten by the dhcpd daemon, when the daemon creates a regular file (object class file) in a directory with the type dhcpd_state_t, the file is automatically assigned the file context of dhcp_state_t. This allows the dhcpd daemon to fully control just the DHCP lease files in /var/lib/dhcp/ and not, for example, the dhclient files in the same directory.

This shows a case where the policy explicitly does not want a file to gain the default label from the parent directory. To prevent this, a type_transition is put into place to guide the context when the file is created.

dhcpd_tmp_t

There are several direct rules and transitions for dhcpd_tmp_t, and multiple indirect rules through the attribute file_type.

These rules describe how dhcpd_t can act upon an object of the type dhcpd_tmp_t, which is the type of the dhcpd temporary files in /tmp/. For example, dhcpd_t can create, read, and get and set file attributes on files, socket files, and FIFO files that have the type dhcpd_tmp_t. Similar actions can be done with directories (dir) and file linking (lnk_file):

```
allow dhcpd_t dhcpd_tmp_t : { file sock_file fifo_file } \
  { create ioctl read getattr lock write setattr append link \
  unlink rename };
allow dhcpd_t dhcpd_tmp_t : lnk_file { create read getattr \
  setattr link unlink rename };
allow dhcpd_t dhcpd_tmp_t : dir { create read getattr lock \
  setattr ioctl link unlink rename search add_name \
  remove_name reparent write rmdir };
```

Having this separate derived type isolates the dhcpd temporary files to ensure that only dhcpd can read and write these files, and not any other daemon. Similarly, other temporary files are protected by being in their own type that dhcpd cannot access. For example, this protects the daemon from using a malicious symlink in /tmp/.

These rules enable the dhcpd daemon to create its files and directories in /tmp. The first rule specifies that when the dhcpd_t domain creates a file in a directory with the type tmp_t, the new subdirectory should be labeled with the dhcpd_tmp_t type. Similarly, the second rule specifics the same transition for a file, file link, socket file, or FIFO (named pipe):

```
type_transition dhcpd_t tmp_t : dir dhcpd_tmp_t;
type_transition dhcpd_t tmp_t : { file lnk_file sock_file \
  fifo_file } dhcpd_tmp_t;
```

The indirect rules are derived from rules associated with the file_type attribute. These deal with allowing file systems to associate default file types, and the manipulation of file_type objects such as dhcpd_tmp_t by the unconfined_t domain:

```
allow { file_type device_type }  fs_t : filesystem associate;
allow file_type removable_t : filesystem associate;
allow file_type nfs_t : filesystem associate;
allow unconfined_t file_type : filesystem  *;
allow unconfined_t file_type : { dir file lnk_file sock_file \
  fifo_file chr_file blk_file } *;
allow unconfined_t file_type : { unix_stream_socket \
  unix_dgram_socket } name_bind;
```

The dhcpd_tmp_t type is also influenced by two generic neverallow assertions. Assertions are discussed in Section 2.8 *TE Rules - Access Vectors*.

dhcpd_var_run_t

This security context is also part of the file_type attribute and shares those rules with dhcpd_tmp_t and others. The direct rules that govern dhcpd_var_run_t allow the dhcpd_t domain to manipulate files and directories with the dhcpd_var_run_t type in the /var/run/ file system. This is the directory where process IDs exist, and this rule allows for the creation and manipulation of /var/run/dhcpd.pid :

```
allow dhcpd_t dhcpd_var_run_t : file  { create ioctl read \
  getattr lock write setattr append link unlink rename };
```

```
allow dhcpd_t dhcpd_var_run_t : dir  { read getattr lock \
  search ioctl add_name remove_name write };
type_transition dhcpd_t var_run_t : file dhcpd_var_run_t;
```

dhcp_etc_t

The two direct rules using this type allow the dhcpd_t domain to read and get attributes on files of the type dhcp_etc_t, as well as search directories of the same type. This means the daemon cannot overwrite the configuration file. Indirect rules derive from the dhcp_etc_t type being part of the file_type attribute set, along with dhcpd_tmp_t, dhcpd_var_run_t, and others:

```
allow dhcpd_t dhcp_etc_t : file { read getattr };
allow dhcpd_t dhcp_etc_t : dir search;
```

dhcp_state_t

In addition to being covered by the rules governing the file_type attribute, this file type has two direct policy rules. The first allows the dhcpd_t domain to perform standard file system functions, such as read, write, and lock on directories, with the type dhcp_state_t. This directory is defined as /var/lib/dhcp/ in $SELINUX_SRC/file_contexts/program/dhcpc.fc, and is where dhcpd stores its lease files. The second rule is the transition rule stating when the dhcpd_t domain creates a file in a directory labeled dhcp_state_t, this file gets a security type of dhcpd_state_t:

```
allow dhcpd_t dhcp_state_t : dir { read getattr lock search \
 ioctl add_name remove_name write };
type_transition dhcpd_t dhcp_state_t : file dhcpd_state_t;
```

**Note**

There are two distinct security contexts being discussed here: dhcp_state_t and dhcpd_state_t.

dhcp_state_t is the type of the directory /var/lib/dhcp where both dhcpd and other clients and daemons store DHCP lease information.

dhcpd_state_t is the type in the security label of a DHCP lease file created in /var/lib/dhcp by the dhcpd daemon, running in the domain of dhcpd_t.

The dhcpd_state_t type is a derivative type, the way dhcpc_state_t derives from the dhcpc_t domain in a stricter policy.

In the /var/lib/dhcp directory, the only allowed actions of the dhcpd_t domain are a series of directory-level operations. The domain cannot affect the files within unless those files are of the type dhcpd_state_t:

```
allow dhcpd_t dhcpd_state_t:file { create ioctl read getattr lock \
  write setattr append link unlink rename };
```

This separation allows the different DHCP applications to keep lease information in the same, traditional directory, yet not be able to affect other DHCP program files.

## 4.3. Boolean Values for dhcpd

SELinux has one Boolean for dhcpd: dhcpd_disable_trans. This is the standard Boolean for all targeted daemons, allowing you to disable the transition from unconfined_t to dhcpd_t. The value of this Boolean is set to false by default.

This Boolean can be changed via the **system-config-securitylevel** application or the /usr/sbin/setsebool -P dhcpd_disable_trans 1 command.

Booleans are explained in Section 3.2 *Files and Directories of the Targeted Policy*.

# II. Working With SELinux

This part discusses how to work with SELinux.

## Table of Contents

# Chapter 5.

# Controlling and Maintaining SELinux

SELinux presents both a new security paradigm and a new set of practices and tools for administrators and some end-users. The tools and techniques discussed in this chapter focus on standard operations performed by administrators, end-users, and analysts. More complex operations, such as compiling a policy after a local change, are covered in Chapter 7 *Compiling SELinux Policy*.

## 5.1. End User Control of SELinux

In general, end users have little interaction with SELinux when Red Hat Enterprise Linux is running the targeted policy. This is because users are running in the domain of `unconfined_t` along with the rest of the system *except* the targeted daemons. This means that when you as an end-user come across a need to use a special SELinux tool or even to check and change the context for a file, it is likely to be when you are working with one of the targeted daemons. You can read more about the targeted daemons in Section 3.1 *What is the Targeted Policy?*.

In most situations, standard DAC controls stop you from doing what you are not permitted before you are stopped by SELinux, and you'll never generate an `avc: denied` message.

These sections cover the general tasks and practices that an end-user might need to do on Red Hat Enterprise Linux. Users of all privilege levels need to do these tasks as well.

### 5.1.1. Move or Copy Files

In file system operations, security context must now be considered in terms of the label of the file, the process touching it, and the directories where the operation is happening. Because of this, moving and copying files with `mv` and `cp` may have unexpected results.

Unless you tell it otherwise, `cp` follows the default behavior of creating a new file based on the domain of the creating process and the type of the target directory. Unless there is a specific rule setting the label, the file inherits the type from the target directory. The `-Z` *`user:role:type`* option allows you to specify what label you want the new file to have.

```
touch bar foo
ls -Z bar foo
-rw-rw-r--  auser   auser   user_u:object_r:user_home_t   bar
-rw-rw-r--  auser   auser   user_u:object_r:user_home_t   foo

# Doing a cp creates a file in the new location with the default
# type based on the creating process and target directory.  In
# this case, there not being a specific rule about cp and /tmp,
# the new file has the type of the parent directory:

cp bar /tmp
ls -Z /tmp/bar
-rw-rw-r--  auser   auser   user_u:object_r:tmp_t   /tmp/bar

# The -Z option allows you to specify the label for the new file:

cp -Z user_u:object_r:user_home_t foo /tmp
ls -Z /tmp/foo
-rw-rw-r--  auser   auser   user_u:object_r:user_home_t   /tmp/foo
```

The type `tmp_t` is the default type for temporary files.

Moving files with mv retains the type the file started with. This may cause problems, for example, if you move files with the type user_home_t into ~/public_html, httpd is not able to serve them until you relabel the file. You can read about file relabeling in Section 5.1.3 *Relabel a File or Directory's Security Context*.

| Command | Behavior |
|---|---|
| mv | The file retains its original label. This may cause problems, confusion, or minor insecurity. For example, the program tmpwatch running in the domain sbin_t might not be allowed to delete an aged file in /tmp because of the file's type. |
| cp | A plain copy creates the new file following the default behavior based on the domain of the creating process (cp) and the type of the target directory. |
| cp -Z user:role:type | The new file is relabeled as it is created based on the command line option. The extended GNU option --context is the same as -Z. |

**Table 5-1. Behavior of mv and cp**

## 5.1.2. Check the Security Context of a Process, User, or File Object

In Red Hat Enterprise Linux, the -Z option is equivalent to --context, and can be used with ps, id, ls, and cp, which is explained in Table 5-1.

The ps command can create a lot of output, so this example is showing only a small sample. Most of the processes are running in unconfined_t, with a few exceptions. You can tell a process started from a root login by the role setting on the label, for example with one of the bash processes:

```
ps -Z
LABEL                             PID TTY          TIME CMD
user_u:system_r:unconfined_t    18543 pts/7    00:00:00 bash
user_u:system_r:unconfined_t    22846 pts/7    00:00:00 ps
ps -eZ
...
user_u:system_r:unconfined_t     1041 ?        00:00:00 udevd
user_u:system_r:unconfined_t     1511 ?        00:00:00 kjournald
user_u:system_r:unconfined_t     1512 ?        00:00:00 kjournald
user_u:system_r:syslogd_t        1873 ?        00:00:01 syslogd
user_u:system_r:unconfined_t     1877 ?        00:00:00 klogd
user_u:system_r:unconfined_t     1888 ?        00:00:34 irqbalance
user_u:system_r:portmap_t        1899 ?        00:00:00 portmap
user_u:system_r:unconfined_t     1919 ?        00:00:00 rpc.statd
user_u:system_r:unconfined_t     1952 ?        00:00:00 rpc.idmapd
...
user_u:system_r:unconfined_t    17252 ?        00:00:01 sshd
root:system_r:unconfined_t      17254 pts/1    00:00:00 bash
user_u:system_r:unconfined_t    17390 ?        00:00:04 gconfd-2
...
user_u:system_r:unconfined_t     1160 ?        00:00:00 firefox
user_u:system_r:unconfined_t     1541 ?        00:00:00 \
  run-mozilla.sh
user_u:system_r:unconfined_t     1558 ?        00:01:37 firefox-bin
```

For id, the -Z option is only usable by itself, it cannot be combined with other options. In this example, the change to root using su did not cause a change in role. In a stricter policy, su is capable of making a role change as well, i.e., from system_r to sysadm_r. This removes the step of using newrole following a su command:

```
# You are an ordinary user here:

whoami
auser
id -Z
user_u:system_r:unconfined_t

# Switching to root changes your UID:

su - root
Password:
whoami
root

# Only the SELinux user name changed, which has no effect in
# the targeted policy.
id -Z
root:system_r:unconfined_t
```

Using the -Z option with ls groups together common long format information. The display choices focus on what you might want when considering the security permissions of a file. It displays mode, user, group, security context, and file name.

```
cd /etc
ls -Z h* -d
drwxr-xr-x  root root  system_u:object_r:etc_t         hal
-rw-r--r--  root root  system_u:object_r:etc_t         host.conf
-rw-r--r--  root root  user_u:object_r:etc_t           hosts
-rw-r--r--  root root  system_u:object_r:etc_t         hosts.allow
-rw-r--r--  root root  system_u:object_r:etc_t         hosts.canna
-rw-r--r--  root root  system_u:object_r:etc_t         hosts.deny
drwxr-xr-x  root root  system_u:object_r:hotplug_etc_t  hotplug
drwxr-xr-x  root root  system_u:object_r:etc_t         hotplug.d
drwxr-xr-x  root root  system_u:object_r:httpd_sys_content_t htdig
drwxr-xr-x  root root  system_u:object_r:httpd_config_t httpd
```

## 5.1.3. Relabel a File or Directory's Security Context

You may need to relabel a file when moving or copying into special directories related to the targeted daemons, such as ~/public_html directories, or when writing scripts that work in directories outside of /home.

There are two general kinds of relabeling operations, one where you are deliberately changing the type of a file, the other where you are restoring files to the default state according to policy. There are also relabeling operations that an administrator performs, and those are covered in Section 5.2.2 *Relabel a File System*.

**Tip**

Since most of SELinux permission control in the targeted policy is type enforcement, you can primarily ignore the user and role information in a security label and focus on just changing the type. This saves you some keystrokes, and keeps you from worrying about the roles and users settings on your files.

**Note**

If relabeling affects the label on a daemon's executable, you want to restart the daemon to be sure it is running in the correct domain. For example, if your `/usr/sbin/mysqld` has the wrong security label and this is fixed by a relabeling operation such as `restorecon`, you must restart `mysqld` after the relabeling. The executable file having the proper type of `mysqld_exec_t` ensures it transitions into the proper domain when started.

Use `chcon` when you have a file that is not the type you want it to be. You must know the new type you want instead:

```
# These directories and files are labeled with the default type
# defined for file system objects created in /home:

cd ~
ls -Zd public_html/
drwxrwxr-x  auser  auser  user_u:object_r:user_home_t public_html/
ls -Z web_files/
-rw-rw-r--  auser  auser  user_u:object_r:user_home_t   1.html
-rw-rw-r--  auser  auser  user_u:object_r:user_home_t   2.html
-rw-rw-r--  auser  auser  user_u:object_r:user_home_t   3.html
-rw-rw-r--  auser  auser  user_u:object_r:user_home_t   4.html
-rw-rw-r--  auser  auser  user_u:object_r:user_home_t   5.html
-rw-rw-r--  auser  auser  user_u:object_r:user_home_t   index.html

mv web_files/* public_html/
ls -Z public_html/
-rw-rw-r--  auser  auser  user_u:object_r:user_home_t     1.html
...

# If you want to make these files viewable from a special user
# public HTML folder, they need to have a type that httpd has
# permissions to read, presuming the Apache HTTP server is configured
# for UserDir and the Boolean value httpd_enable_homedirs is
# enabled.

chcon -R -t httpd_used_content_t public_html/
ls -Z public_html
-rw-rw-r--  auser  auser  user_u:object_r:httpd_user_content_t \
  1.html
...

ls -Z public_html/ -d
drwxrwxr-x  auser  auser  user_u:object_r:httpd_user_content_t \
public_html/
```

**Tip**

> If the file has no label, such as a file created while SELinux was disabled in the kernel, you need to give it a full label with `chcon system_u:object_r:shlib_t foo.so`. If you don't, you get an error about applying a partial context to an unlabeled file.

Use `restorecon` when you want to restore files to the policy default. There are two other methods to do this that work on the entire file system, `fixfiles` or a policy relabeling operation. These require you to be the root user. Cautions against both of these methods appear in Section 5.2.2 *Relabel a File System*.

This example shows restoring the default user home directory context to a set of files that have different types:

```
# These two sets of files have different types, and are
# being moved into a directory for archiving.  Their contexts
# are different from each other, and incorrect for a standard
# user's home directory:

ls -Z /tmp/{1,2,3}
-rw-rw-r--  auser  auser  user_u:object_r:tmp_t            /tmp/1
-rw-rw-r--  auser  auser  user_u:object_r:tmp_t            /tmp/2
-rw-rw-r--  auser  auser  user_u:object_r:tmp_t            /tmp/3
mv /tmp/{1,2,3} archives/
mv public_html/* archives/
ls -Z archives/
-rw-rw-r--  auser  auser  user_u:object_r:tmp_t            1
-rw-rw-r--  auser  auser  user_u:object_r:httpd_user_content_t \
  1.html
-rw-rw-r--  auser  auser  user_u:object_r:tmp_t            2
-rw-rw-r--  auser  auser  user_u:object_r:httpd_user_content_t \
  2.html
-rw-rw-r--  auser  auser  user_u:object_r:tmp_t            3
-rw-rw-r--  auser  auser  user_u:object_r:httpd_user_content_t \
  3.html
-rw-rw-r--  auser  auser  user_u:object_r:httpd_user_content_t \
  4.html
-rw-rw-r--  auser  auser  user_u:object_r:httpd_user_content_t \
  5.html
-rw-rw-r--  auser  auser  user_u:object_r:httpd_user_content_t \
index.html

# The directory archives/ is already the default type
# because it was created in the user's ~/ directory:

ls -Zd archives/
drwxrwxr-x  auser  auser  user_u:object_r:user_home_t  archives/

# Relabeling with restorecon uses the default file contexts set
# by the policy, so these files are labeled with the default
# label for the directory they are in.

/sbin/restorecon -R archives/
ls -Z archives/
-rw-rw-r--  auser  auser  system_u:object_r:user_home_t    1
-rw-rw-r--  auser  auser  system_u:object_r:user_home_t    1.html
-rw-rw-r--  auser  auser  system_u:object_r:user_home_t    2
-rw-rw-r--  auser  auser  system_u:object_r:user_home_t    2.html
-rw-rw-r--  auser  auser  system_u:object_r:user_home_t    3
-rw-rw-r--  auser  auser  system_u:object_r:user_home_t    3.html
```

```
-rw-rw-r--  auser  auser  system_u:object_r:user_home_t    4.html
-rw-rw-r--  auser  auser  system_u:object_r:user_home_t    5.html
-rw-rw-r--  auser  auser  system_u:object_r:user_home_t    \
  index.html
```

### 5.1.4. Make Backups or Archives That Retain Security Contexts

The `tar` utility does not yet support archiving and restoring extended attributes in Red Hat Enterprise Linux 4. Instead, you can do this using the `star` utility, with the appropriate options `-xattr` and `-H=exustar`. This ensures that extra attributes are captured and the header for the `*.star` file is of a type that fully supports xattrs:

```
# Note how the two directories have different labels.
# The ellipses '...' cover the unimportant part of the
# file context for printing purposes:

ls -Z public_html/ web_files/
public_html/:
-rw-rw-r--  auser  auser  ...httpd_user_content_t 1.html
-rw-rw-r--  auser  auser  ...httpd_user_content_t 2.html
-rw-rw-r--  auser  auser  ...httpd_user_content_t 3.html
-rw-rw-r--  auser  auser  ...httpd_user_content_t 4.html
-rw-rw-r--  auser  auser  ...httpd_user_content_t 5.html
-rw-rw-r--  auser  auser  ...httpd_user_content_t index.html

web_files/:
-rw-rw-r--  auser  auser  user_u:object_r:user_home_t  1.html
-rw-rw-r--  auser  auser  user_u:object_r:user_home_t  2.html
-rw-rw-r--  auser  auser  user_u:object_r:user_home_t  3.html
-rw-rw-r--  auser  auser  user_u:object_r:user_home_t  4.html
-rw-rw-r--  auser  auser  user_u:object_r:user_home_t  5.html
-rw-rw-r--  auser  auser  user_u:object_r:user_home_t  index.html

star -xattr -H=exustar -c -f all_web.star public_html/ web_files/
star: 11 blocks + 0 bytes (total of 112640 bytes = 110.00k).

ls -Z all_web.star
-rw-rw-r--  auser  auser  user_u:object_r:user_home_t \
 all_web.star
cp all_web.star /tmp/
cd /tmp/

# Here in /tmp, if there is no specific policy to make a derivative
# temporary type, the default behavior is to acquire the tmp_t type
# for new files, such as the newly copied file all_web.star,

ls -Z all_web.star
-rw-rw-r--  auser  auser  user_u:object_r:tmp_t  all_web.star

# *.star files are usable by tar, but tar does not know how to
# extract extended attributes.  Without a label on the file,
# the creation of new files in /tmp again chooses the default file
# type of tmp_t:

tar -xvf all_web.star
...
ls -Z /tmp/public_html/ /tmp/web_files/
/tmp/public_html/:
-rw-rw-r--  auser  auser  user_u:object_r:tmp_t  1.html
-rw-rw-r--  auser  auser  user_u:object_r:tmp_t  2.html
```

```
-rw-rw-r--  auser  auser  user_u:object_r:tmp_t  3.html
-rw-rw-r--  auser  auser  user_u:object_r:tmp_t  4.html
-rw-rw-r--  auser  auser  user_u:object_r:tmp_t  5.html
-rw-rw-r--  auser  auser  user_u:object_r:tmp_t  index.html

/tmp/web_files/:
-rw-rw-r--  auser  auser  user_u:object_r:tmp_t  1.html
-rw-rw-r--  auser  auser  user_u:object_r:tmp_t  2.html
-rw-rw-r--  auser  auser  user_u:object_r:tmp_t  3.html
-rw-rw-r--  auser  auser  user_u:object_r:tmp_t  4.html
-rw-rw-r--  auser  auser  user_u:object_r:tmp_t  5.html
-rw-rw-r--  auser  auser  user_u:object_r:tmp_t  index.html

rm -rf /tmp/public_html/ /tmp/web_files/

# Now you can expand the archives using star and it
# restores the extended attributes:

star -xattr -x -f all_web.star
star: 11 blocks + 0 bytes (total of 112640 bytes = 110.00k).
ls -Z /tmp/public_html/ /tmp/web_files/
/tmp/public_html/:
-rw-rw-r--  auser  auser  ...httpd_sys_content_t 1.html
-rw-rw-r--  auser  auser  ...httpd_sys_content_t 2.html
-rw-rw-r--  auser  auser  ...httpd_sys_content_t 3.html
-rw-rw-r--  auser  auser  ...httpd_sys_content_t 4.html
-rw-rw-r--  auser  auser  ...httpd_sys_content_t 5.html
-rw-rw-r--  auser  auser  ...httpd_sys_content_t index.html

/tmp/web_files/:
-rw-rw-r--  auser  auser  user_u:object_r:user_home_t  1.html
-rw-rw-r--  auser  auser  user_u:object_r:user_home_t  2.html
-rw-rw-r--  auser  auser  user_u:object_r:user_home_t  3.html
-rw-rw-r--  auser  auser  user_u:object_r:user_home_t  4.html
-rw-rw-r--  auser  auser  user_u:object_r:user_home_t  5.html
-rw-rw-r--  auser  auser  user_u:object_r:user_home_t  \
index.html
```

### Caution

If you use an absolute path when you create an archive using `star`, the archive expands on that same path. For example, an archive made with this command restores the files to `/var/log/httpd/`:

```
star -xattr -H=exustar -c -f httpd_logs.star /var/log/httpd/
```

If you attempt to expand this archive, `star` issues a warning if the files in the path are newer than the ones in the archive.

## 5.2. Administrator Control of SELinux

Administrators can expect to do most of the same things that users do in Section 5.1 *End User Control of SELinux*, plus a number of additional tasks that are usually done only at the root level. Using the targeted policy makes tasks measurably easier for the administrator. For example, there is no need to consider adding, editing, or deleting Linux users from the SELinux users, nor do you need to consider roles.

This section covers the types of tasks that an administrator needs to do to maintain Red Hat Enterprise Linux running SELinux.

### 5.2.1. View the Status of SELinux

The command `sestatus` provides a configurable view into the status of SELinux. By itself, the command shows the enabled status, selinuxfs mount point, current enforcing mode and what that is set to in the configuration file, and the policy name and its version number. Following that are a list of all the policy Booleans and their status:

```
/usr/bin/sestatus

SELinux status:         enabled
SELinuxfs mount:        /selinux
Current mode:           enforcing
Mode from config file:  enforcing
Policy version:         18
Policy from config file:targeted

Policy booleans:
allow_ypbind            active
dhcpd_disable_trans     active
httpd_disable_trans     inactive
httpd_enable_cgi        active
...
```

The `-v` option adds on a report about the security contexts of a series of files that are specified in `/etc/sestatus.conf`:

```
Process contexts:
Current context:        root:system_r:unconfined_t
Init context:           user_u:system_r:unconfined_t
/sbin/mingetty          user_u:system_r:unconfined_t
/usr/sbin/sshd          user_u:system_r:unconfined_t

File contexts:
Controlling term:       root:object_r:devpts_t
/etc/passwd             system_u:object_r:etc_t
/etc/shadow             system_u:object_r:shadow_t
/bin/bash               system_u:object_r:shell_exec_t
/bin/login              system_u:object_r:bin_t
...
```

### 5.2.2. Relabel a File System

You may never need to relabel an entire file system. This usually occurs only when labeling a file system for SELinux for the first time, or when switching between different kinds of policy, such as going from the targeted to the strict policy.

There is one good method for relabeling the file system. You may also hear about two other methods, both of which are *not recommended*. Here they are in order:

1. The best and cleanest method to relabel is to let `init` do it for you on boot.

   ```
   touch /.autorelabel
   reboot
   ```

   By allowing the relabeling to occur early in the reboot process, you ensure that applications have the right labels when they are started and that they are started in the right order. If you relabel a live file system without rebooting, you may have processes running under the incorrect context. Making sure all the daemons are restarted and running in the right context can be difficult.

2. It is possible to relabel a live file system using `fixfiles`, or to relabel based on the RPM database:

   ```
   fixfiles relabel
   fixfiles -R packagename restore
   ```

   Using the ability of `fixfiles` to restore contexts from packages is safer and quicker.

   **Caution**

   Running `fixfiles` on the whole file system without rebooting may make the system unstable.

   If the relabeling operation applies a new policy that is different from the policy that was in place when the system booted, existing processes may be running in incorrect and insecure domains. For example, a process could be in a domain that is not an allowed transition for that process in the new policy, granting unexpected permissions to that process alone.

   In addition, one of the options to `fixfiles relabel` prompts for approval to empty `/tmp/` because it is not possible to reliably relabel `/tmp/`. Since `fixfiles` is run as root, temporary files that applications are relying upon are erased. This could make the system unstable or behave unexpectedly.

3. There is another method using the source policy. You want to avoid `make relabel` for the same reason you avoid using `fixfiles`.

## 5.2.3. Managing NFS Home Directories

In Red Hat Enterprise Linux 4 most targeted daemons do not interact with user data and are not affected by NFS-mounted home directories. One exception is Apache HTTP. For example, CGI scripts that are on the mounted file system have the `nfs_t` type, which is not a type `httpd_t` is allowed to execute.

If you are having problems with the default type of `nfs_t`, try mounting the home directories with a different context:

```
mount -t nfs -o context=user_u:object_r:user_home_dir_t \
 fileserver.example.com:/shared/homes/ /home
```

**Caution**

Section 5.2.13 *Specifying the Security Context of Entire File Systems* explains how to mount a directory so that `httpd` is allowed to execute scripts. Doing that for user home directories gives Apache HTTP increased access to those directories. Remember that a mountpoint label is for the entire mounted file system.

Future versions of the SELinux policy address the functionality of NFS.

## 5.2.4. Grant Access to a Directory or a Tree

Just as with regular Linux DAC permissions, a targeted daemon must have SELinux permissions to be able to descend the directory tree from the root. This does not mean that a directory and its contents need to have the same type. There are many types, such as `root_t`, `tmp_t`, and `usr_t` that grant read access for a directory. These are good types to use if you have a directory with no secret information you want to be widely readable. It might also make a good directory type for a parent directory of more secured directories with different contexts.

If you are working with an `avc: denied` message, there are some common problems that arise with directory traversal. For example, many programs do an equivalent command to `ls -l /` that is not necessary to their operation but generates a denial message in the logs. For this you need to create a `dontaudit` rule in your `local.te` file. Read more about this in Chapter 8 *Customizing and Writing Policy*.

When you are interpreting the AVC denial message, you might get misled by the `path=/` component. This path is not related to the label for the root file system, `/`. It is actually relative to the root of the file system on the device node. For example, if your `/var/` directory is located on an LVM (*Logical Volume Management*[1]) device, `/dev/dm-0`, the device node is identified in the message as `dev=dm-0`. When you see `path=/` in this example, that is the top level of the LVM device `dm-0`, not neccesarily the same as the root file system designation `/`.

## 5.2.5. Load a Policy

There are two routes to loading a policy. One is to install a binary policy from a package or copy a custom binary policy into $SELINUX_POLICY/. The other is to use the policy source and load eithr the supported or a custom policy. For information on this second option, read Chapter 7 *Compiling SELinux Policy* and Chapter 8 *Customizing and Writing Policy*.

**Note**

> It is not common to install the policy sources unless you need to work with them directly. On a normal production server, you are not likely to have policy source installed even if you are running a customized policy. You develop that policy on a separate machine that has the source installed, and deploy it as a binary policy to production machines.

You can upgrade the package using `up2date` or `rpm`. If you are managing your own custom policy, either package it or copy the binary policy file `policy.`*`XY`* to the target machine.

However, if you have the policy source package installed *and* you have loaded the policy from source, such as running `make load` or `make reload` in the `$SELINUX_SRC/` directory, then installing binary policy packages is slightly more complicated.

The install scripts packaged with the policy check to see if you have the policy source package installed and if you loaded policy from source. It does this by comparing the file at `$SELINUX_POLICY/policy.`*`XY`* with the binary policy from the package. If they are different, the new binary policy is created with an `.rpmnew` file extension. This way you are protected from having your customizations overwritten by a policy upgrade.

If you want to use the binary policy, move the replacement over the older version:

```
rpm -Uvh /tmp/selinux-policy-targeted-*
Preparing...                ########################### [100%]
   1:selinux-policy-targeted########################### [ 50%]
```

[1] LVM is the grouping of physical storage into virtual pools that are partitioned into logical volumes.

```
warning: /etc/selinux/targeted/policy/policy.18 created as \
  /etc/selinux/targeted/policy/policy.18.rpmnew
   2:selinux-policy-targeted########################## [100%]

mv /etc/selinux/targeted/policy/policy.18.rpmnew \
   /etc/selinux/targeted/policy/policy.18
```

Otherwise, install the new policy source and load a new policy.

This situation occurs as a protection against an updated policy package overwriting a custom binary policy. Future policy packages will address this challenge further.

If you want to deploy a custom binary policy, read Section 8.4 *Deploying Customized Binary Policy*.

### 5.2.6. Backup and Restore the System

Refer to the explanation in Section 5.1.4 *Make Backups or Archives That Retain Security Contexts*.

### 5.2.7. Enable or Disable Enforcement

You can enable and disable SELinux enforcement in runtime or configure it for system boot, using the command line or GUI. There are three modes for SELinux to be in: *disabled*, meaning not enabled in the kernel; *permissive*, meaning SELinux is running and logging but not controlling permissions; *enforcing*, meaning SELinux is running and enforcing policy.

To toggle enforcement during runtime, use the `setenforce [ 0 | 1 ]` command. The `0` option turns enforcement off, the `1` option turns it on.

```
# sestatus informs you of the two permission mode statuses,
# the current mode in runtime and the mode from the config
# file referenced during boot:

sestatus | grep -i mode
Current mode:           permissive
Mode from config file:  permissive

# Changing the runtime enforcement doesn't effect the
# boot time configuration:

setenforce 1
sestatus | grep -i mode
Current mode:           enforcing
Mode from config file:  permissive
```

However, you may be looking for something more subtle. For example, if you are having trouble with `named` and SELinux, you can turn off enforcing for just that daemon:

```
# This gets the current status of the Boolean:

getsebool named_disable_trans
named_disable_trans --> inactive

# This sets the runtime value only.  To flush the pending
# value to disk use the -P option.

setsebool named_disable_trans 1
getsebool named_disable_trans
named_disable_trans --> active
```

You can configure all of these settings using **system-config-securitylevel**. The same configuration files are used, so changes show up bidirectionally.

- To set SELinux to enforcing, choose the **SELinux** tab and select the checkboxes next to **Enabled (Modification Requires Reboot)** and **Enforcing**. After clicking **OK**, you need to reboot if you have just enabled SELinux from disabled.
- To set SELinux to permissive mode, deselect the checkbox next to **Enforcing**. The mode changes when you click the **OK** button.
- To disable SELinux enforcement over a targeted daemon, you are setting a Boolean value so that SELinux does not transition the program to the targeted domain.

  In the **SELinux** tab, under the **Modify SELinux Policy** section, there is a menu **SELinux Service Protection**. Clicking on the triangle opens that menu, where you can choose to **Disable SELinux protection for *foo* daemon**. Clicking the **OK** button makes the change take effect.

If you are interested in controlling these configurables with scripts, the tools `setenforce(1)`, `getenforce(1)`, and `selinuxenabled(1)` may be useful to you.

## 5.2.8. Change a Boolean Setting

Booleans are reconfigurable in runtime, and you can choose to write the setting to the configuration files for the next policy load.

The reliable command line method is to use `setsebool`:

```
setsebool httpd_enable_homedirs 1
```

By itself, `setsebool` only changes the current state of the Booleans. The `-P` option writes all pending changes to the file `/etc/selinux/targeted/booleans`. In this example you are enabling policy enforcement for a list of daemons:

```
# Any *_disable_trans set to 1 are invoking the conditional that
# prevents the process from transitioning to the domain on exec:

grep disable /etc/selinux/targeted/booleans | grep 1
httpd_disable_trans=1
mysqld_disable_trans=1
ntpd_disable_trans=1

# You can pass any number of boolean_value=0|1
setsebool -P httpd_disable_trans=0 mysqld_disable_trans=0 \
  ntpd_disable_trans=0
grep disable booleans | grep 1
```

If you already know the setting of a Boolean, you can use `togglesebool` *`boolean_name`* to flip the setting.

Using **system-config-securitylevel**, Boolean control is in the **SELinux** tab, under the **Modify SELinux Policy** section. Each Boolean has a checkbox in the menu. Settings take effect when you click **OK**.

## 5.2.9. Enable or Disable SELinux

**Important**

> Changes you make to files while SELinux is disabled may give them an unexpected security label, and new files do not have a label. You may need to relabel part or all of the file system after enabling SELinux again.

From the command line, you can edit the file `/etc/sysconfig/selinux`. You'll notice the file is a symlink to `/etc/selinux/config`. The configuration file is self-explanatory. Changing the value of *`SELINUX=`* or *`SELINUXTYPE=`* changes the state of SELinux and the name of the policy to be used upon the next system boot.

```
# This file controls the state of SELinux on the system.
# SELINUX= can take one of these three values:
#       enforcing - SELinux security policy is enforced.
#       permissive - SELinux prints warnings instead of enforcing.
#       disabled - SELinux is fully disabled.
SELINUX=enforcing
# SELINUXTYPE= type of policy in use. Possible values are:
#       targeted - Only targeted network daemons are protected.
#       strict - Full SELinux protection.
SELINUXTYPE=targeted
```

Using **system-config-securitylevel** in the **SELinux** tab, uncheck **Enabled (Modification Requires Reboot)**, click **OK** to accept the changes, then reboot. This immediately changes the setting in `/etc/sysconfig/selinux`.

## 5.2.10. Change the Policy

If you are interested in customizing the policy, read Chapter 8 *Customizing and Writing Policy*. If you have a different policy that you wish to load on your system, such as a strict or other specialized policy, you only need to set *`SELINUXTYPE=policyname`*, where *`policyname`* is the same as the directory `/etc/selinux/`*`policyname`*. This presumes you have the custom policy installed, which is also covered in Chapter 8 *Customizing and Writing Policy*, as well as troubleshooting steps to get a custom policy working on a different system. After changing the *`SELINUXTYPE`* parameter, you want to `touch /.autorelabel` and reboot the system.

To use **system-config-securitylevel** to switch the policy, in the **SELinux** tab there is a drop-down menu **Policy Type:**. It is set to **targeted** and your custom policy appears there as **policyname** once the directory structure is in `/etc/selinux`. Click **OK** to accept the changes and reboot the system.

## 5.2.11. Troubleshoot User Problems With SELinux

This presents a brief methodology for troubleshooting problems that your users might have with SELinux.

1. Deciphering the denial message is the first step in troubleshooting. Read Section 2.8.1 *Understanding an `avc: denied` Message* for how to do that. You might want to use **seaudit** if there are a large number of AVC audit messages. You can read more about **seaudit** in Section 6.2 *Using **seaudit** for Audit Log Analysis*. Here are the questions you want answered:
   - What is the process that is being blocked? You can find its context from the `scontext=` portion of the message.

- What is the target object? The path= and the tclass= tell you where and what the object is. You get it's context from tcontext=. You may need the ino= to find an object if it's path is not evident. This may happen because SELinux reports the path as relative to the device node dev=.
- What is the permission attempted?

2. Knowing these essential who, what, where, and how questions should help you in determining the why. At this point it may be obvious, such as the tcontext= being set to a context the process clearly should not be writing to. This may point back to troubles in the application or script, or troubles in the type for the subject or object.
3. If you need to analyze the policy further, you can try using the source and target contexts as search parameters with the **apol** tool. You can learn more about how to do this in Section 6.3 *Using **apol** for Policy Analysis*.
4. If you think the interaction should be allowed and represents a policy bug, you can insert policy to allow it. Read Chapter 8 *Customizing and Writing Policy* for information on doing this, and file a bug report at http://bugzilla.redhat.com.

### 5.2.12. Read an `avc: denied` Message

For information on how to read an AVC message, read Section 2.8.1 *Understanding an avc: denied Message*.

### 5.2.13. Specifying the Security Context of Entire File Systems

Using the mount -o context= command you can set a single context for an entire file system. This might be an already mounted file system that supports xattrs, or a network file system that obtains a genfs label such as cifs_t or nfs_t. This is explained in Section 2.4 *File System Security Contexts*

For example, if you need to have Apache HTTP read from a mounted directory or loopback file system, you need to set the type to httpd_sys_content_t:

```
mount -t nfs -o context=system_u:object_r:httpd_sys_content_t \
server1.example.com:/shared/scripts /var/www/cgi
```

**Tip**

When troubleshooting httpd and SELinux problems, reduce the complexity of your situation. For example, if you have the file system mounted at /mnt and then symlinked to /var/www/html/foo, you have two security contexts to be concerned with. Since one is of the object class file and the other lnk_file, they are treated differently by the policy and unexpected behavior may occur.

### 5.2.14. Run a Command in a Specified Security Context

This is useful for scripting or testing policy, although it can be tricky to do correctly. The runcon command lets you specify the domain that you want to run a program or script in. For example, you could runcon -t httpd_t /path/to/script for a script that tested for mislabeled content.

```
# The arguments that appear after the command are considered to
# be part of the command being run
```

```
runcon -t httpd_t ~/bin/contexttest -ARG1 -ARG2

# You can also specify the entire context
runcon user_u:system_r:httpd_t ~/bin/contexttest
```

### 5.2.15. Useful Commands for Scripts

You many need access to SELinux information and capabilities for scripts you write in administrating your system. This is a list of useful commands introduced with SELinux:

`getenforce`

This command returns the enforcing status of SELinux.

`setenforce [ *Enforcing* | *Permissive* | *1* | *0* ]`

This command controls the enforcing mode of SELinux. The option `1` or `Enforcing` tells SELinux to begin enforcing. The option `0` or `Permissive` tells SELinux to stop enforcing, although it continues logging access violations.

`selinuxenabled`

This command exits with a status of `0` if SELinux is enabled, and `-256` if SELinux is disabled.

```
selinuxenabled
echo $?
0
```

`getsebool [-a] [*boolean_name*]`

This command shows the status of all (`-a`) or a specific Boolean can be determined.

`setsebool [-P] boolean_name value | bool1=val1 bool2=val2 ...`

This command sets one or more Boolean values. The option `-P` commits all pending Boolean changes to the configuration file at `/etc/selinux/targeted/booleans`.

`togglesebool boolean ...`

This command toggles the setting of one or more Booleans. Whatever the setting was, it is now switched to the opposite. This effects Boolean settings in memory only, and does not change the Boolean setting in `/etc/selinux/targeted/booleans`.

### 5.2.16. Assume a New Role

This program lets you run a new shell with the specified type and/or role. Switching roles does not have the same meaning in the targeted policy as it does in a strict policy, so that function is largely ignored. It may be useful to you to assume a new type for testing, validation, and development purposes:

```
newrole -r role_r -t type_t [-- [ARGS]...]
```

The `ARGS` following the `--` are passed directly to the shell. The shell chosen is based on the user's entry in `/etc/passwd`.

### 5.2.17. When to Reboot

Your primary reason for rebooting with SELinux is to get your file system properly labeled using the `/.autorelabel` file. Another reason might be to completely enable or disable SELinux.

Otherwise, you can safely make SELinux permissive by using `setenforce 0`.

## 5.3. Analyst Control of SELinux

This section presents some common tasks that a security analyst might need to do on an SELinux system.

### 5.3.1. Enable Kernel Auditing

You may wish to have the full kernel-level auditing available when doing analysis or troubleshooting. This can be quite verbose, since it generates one or more additional audit message(s) for each AVC audit message. To enable, append the parameter *`audit=1`* to your kernel boot line, either through `/etc/grub.conf` or via the GRUB menu during boot.

This is an example of a full audit log entry when `httpd` is denied access to `~/public_html` because the directory is not labeled as Web content:

```
# Notice that the time and serial number stamps in the audit(...)
# field are identical, making it easier to track a specific
# event in the audit logs:

Jan 15 08:03:56 hostname kernel: audit(1105805036.075:2392892): \
avc:  denied  { getattr } for  pid=2239 exe=/usr/sbin/httpd \
path=/home/auser/public_html dev=hdb2 ino=921135 \
scontext=user_u:system_r:httpd_t \
tcontext=system_u:object_r:user_home_t tclass=dir

# This audit message tells more about the source, including the
# kind of syscall involved, showing that httpd tried to stat the
# directory:

Jan 15 08:03:56 hostname kernel: audit(1105805036.075:2392892): \
syscall=195 exit=4294967283 a0=9ef88e0 a1=bfecc0d4 a2=a97ff4 \
a3=bfecc0d4 items=1 pid=2239 loginuid=-1 uid=48 gid=48 euid=48 \
suid=48 fsuid=48 egid=48 sgid=48 fsgid=48

# This message tells more about the target:

Jan 15 08:03:56 hostname kernel: audit(1105805036.075:2392892): \
item=0 name=/home/auser/public_html inode=921135 dev=00:00
```

By design, the serial number stamp is always identical for a particular audited event. The time stamp may not always be identical but most often is identical.

**Note**

If you are using an audit daemon for troubleshooting, the daemon may capture audit messages into another location than `/var/log/messages`, such as `/var/log/audit.log`. Red Hat Enterprise Linux 4 does not ship with an audit daemon, but work on this is ongoing.

### 5.3.2. Dump or View Policy

While there is no formal way to dump the policy in memory, there are several tools which make it easier to view and analyze policy. Here are three ways of viewing the policy.

- The binary policy directory at `$SELINUX_POLICY/` contains information on Booleans and file contexts. You can analyze the binary policy with the `setools` such as **apol** and `seinfo`, which are discussed in Chapter 6 *Tools for Manipulating and Analyzing SELinux*.

  You can read more about where the policy files are located starting in Section 2.2 *Where is the Policy?*.
- For a more thorough analysis, nothing equals the policy source, located in `$SELINUX_SRC/` and discussed extensively in Chapter 2 *SELinux Policy Overview* and Chapter 3 *Targeted Policy Overview*.

  Standard command line text processing tools and the `setools` are two essential methods for viewing and understanding the policy source.
- Currently, the best method for analyzing SELinux policy is to use the `setools`. One GUI tool in particular is **apol**, which provides fairly complex analysis capabilities. This is discussed more thoroughly in Section 6.3 *Using **apol** for Policy Analysis*.

### 5.3.3. Dump and View Logs

The SELinux implementation in Red Hat Enterprise Linux 4 routes AVC audit messages to `/var/log/messages`. You can seek just the audit messages using `grep` and searching for `avc` or `audit`.

As discussed in Section 6.2 *Using **seaudit** for Audit Log Analysis*, `seaudit` is a GUI tool for organizing and analyzing just policy messages. The tool `seaudit-report` generates text or HTML reports of audit messages.

### 5.3.4. Viewing AVC Statistics

The best way to view formatted statistics about the access vector cache is to use `avcstat`. This is explained in Section 6.1 *Information Gathering Tools*.

## 5.4. Policy Writer Control of SELinux

Writing SELinux policy is not a trivial undertaking. The topic cannot easily be covered in a few, simple how-to steps. If you are interested in this topic, read Chapter 7 *Compiling SELinux Policy* and Chapter 8 *Customizing and Writing Policy*. Those chapters contain information on writing, testing, loading, and validating a policy.

# Chapter 6.

# Tools for Manipulating and Analyzing SELinux

An administrator's job may include analyzing and possibly manipulating the SELinux policy, as well as doing performance analysis and tuning. This chapter discusses analysis and tuning.

For policy manipulation, you may wish to support a new daemon or discover and fix a problem, as discussed in Chapter 8 *Customizing and Writing Policy*. One early step to writing policy is analyzing existing policy so that you understand how it works. One example of this is given in Section 2.9.1 *How To Backtrack a Rule*, where a macro is analyzed through the process of backtracking to the source of a set of rules.

While some effective policy analysis can be done using standard command line text manipulation tools, sophisticated policy analysis requires stronger tools. The simpler targeted policy consists of more than 20,000 concatenated lines in `policy.conf`, which is derived from more than 150 macros and thousands of lines of TE rules and file context settings, all interacting in very complex ways. Tools such as **apol** are designed specifically for doing analysis of SELinux policy. This chapter discusses these tools, which are part of the `setools` package. In addition to the GUI analysis tools **seaudit** and **apol**, several command line tools that are useful for gathering information and statistics are explained.

Analysis is also necessary when doing performance tuning. Due to the real and potential workload imposed by the AVC system, you may have some situations where being able to manipulate how this works is useful to improving performance. This chapter presents some methods to tune your SELinux installation.

In order to use these applications, you need both the `setools` and `setools-gui` packages installed. The other packages you need come with the SELinux installation: `libselinux` and `policycoreutils`.

**Tip**

When you are running a privileged application over `ssh`, meaning an application that requires you to have root privileges, you must use the `-Y` option. This option enables trusted X11 forwarding:

```
ssh -Y root@host.example.com
```

The configuration requiring this is enabled by default and is new to Red Hat Enterprise Linux 4.

## 6.1. Information Gathering Tools

These tools are command line tools, providing formatted output. They are harder to use as part of command line piping, but they provide gathered and well formatted information quickly.

`avcstat`

This provides a short output of the access vector cache statistics since boot. You can watch the statistics in real time by specifying a time interval in seconds. This provides updated statistics since the initial output. The statistics file used is `/selinux/avc/cache_stats`, and you can specify a different cache file with the `-f /path/to/file`. For example, this might be useful for reviewing saved snapshots of `/selinux/avc/cache_stats`.

```
avcstat
   lookups       hits    misses     allocs     reclaims     frees
 194658175  194645272     12903      12903          880     12402
```

```
# This shows one second intervals:

avcstat 1
   lookups       hits    misses    allocs   reclaims    frees
 194670327  194657424     12903     12903        880    12402
       493        493         0         0          0        0
       370        370         0         0          0        0
       390        390         0         0          0        0
       366        366         0         0          0        0
       364        364         0         0          0        0

# With these five second intervals, you see the accumulation
# of lookups and hits over the course of the interval.

avcstat 5
   lookups       hits    misses    allocs   reclaims    frees
 194683017  194670114     12903     12903        880    12402
      1966       1966         0         0          0        0
      1824       1824         0         0          0        0
```

The `lookups` field shows the workload of the AVC. It is not uncommon to have the number of `hits` be smaller than the number of `lookups`.

Section 6.4 *Performance Tuning* discusses how to use `avcstat` for performance tuning.

`seinfo`

This utility is useful in describing the break down of a policy, such as the number of classes, types, Booleans, allow rules, and so forth. Similar in function to some aspects of **apol**, `seinfo` is a quick command line utility that takes `policy.conf` or a binary policy file as input.

The results are going to be different between binary and source files. For example, the policy source file uses the `{ }` brackets to group multiple rule elements onto a single line. A similar effect happens with attributes, where a single attribute expands into one or many types. Because these are expanded and no longer relevant in the binary policy file, they have a return value of zero in the search results. However, the number of rules greatly increases as each formerly one line rule using brackets is now a number of individual lines.

Some items are not present in the binary policy. For example, `neverallow` rules are only checked during policy compile, not during runtime, and initial SIDs are not part of the binary policy since they are required prior to the policy being loaded by the kernel during boot.

```
seinfo $SELINUX_SRC/policy.conf

Statistics for policy file: $SELINUX_SRC/policy.conf
Policy Version: v.18
Policy Type: source

   Classes:           53    Permissions:      192
   Types:            317    Attributes:        81
   Users:              3    Roles:              4
   Booleans:          20    Cond. Expr.:       21
   Allow:           2292    Neverallow:         7
   Auditallow:         2    Dontaudit:        225
   Type_trans:        99    Type_change:        0
   Role allow:         5    Role trans:         0
   Initial SIDs:      27

seinfo $SELINUX_POLICY/policy.18

Statistics for policy file: $SELINUX_POLICY/policy.18
Policy Version: v.18
Policy Type: binary
```

```
   Classes:          53    Permissions:     192
   Types:           316    Attributes:        0
   Users:             3    Roles:             4
   Booleans:         20    Cond. Expr.:      21
   Allow:         11134    Neverallow:        0
   Auditallow:        2    Dontaudit:       569
   Type_trans:      157    Type_change:       0
   Role allow:        5    Role trans:        0
   Initial SIDs:      0
```

`sesearch`

Similar to the way that `seinfo` provides light information gathering functionality from **apol** on the command line, `sesearch` lets you search for a particular type in the policy. Policy source or binary can be used.

```
sesearch -a -t httpd_sys_content_t $SELINUX_POLICY/policy.conf

5 Rules match your search criteria
allow  httpd_suexec_t { httpd_sys_content_t \
  httpd_sys_script_ro_t httpd_sys_script_rw_t \
  httpd_sys_script_exec_t } : dir  { getattr search };
allow  httpd_sys_script_t  httpd_sys_content_t : dir  \
  { getattr search };
allow  httpd_t  httpd_sys_content_t : dir  { read getattr \
  lock search ioctl };
allow  httpd_t  httpd_sys_content_t : file  { read getattr \
  lock ioctl };
allow  httpd_t  httpd_sys_content_t : lnk_file  { getattr \
  read };

# This same search, when performed on the binary policy file,
# generates 38 matching rules.
```

There are command line options to `sesearch` to control various factors of the search:

| **Option** | **Behavior** |
|---|---|
| `-s, --source <NAME>` | Search for rules that have the search expression as a source; `<NAME>` is a regular expression. |
| `-t, --target <NAME>` | Search for rules that have `<NAME>` as a target. |
| `-c, --class <NAME>` | Search for rules that have `<NAME>` as the object class. |
| `-p, --perms <P1[,P2...]>` | Search for one or more specific permissions. |
| `--allow` | Search for only `allow` rules. |
| `--neverallow` | Search for only `neverallow` rules. |
| `--audit` | Search for only `dontaudit` and `auditallow` rules. |
| `--type` | Search for only type transition (`type_trans`) and type change (`type_change`) rules. |
| `-i, --indirect` | Do an indirect search, which looks for rules deriving from a type's attribute. |
| `-n, --noregex` | Do not use regular expression matching for types and attributes searched for. |

| Option | Behavior |
|---|---|
| `-a, --all` | Show all rules. You must specify one of the rule types in your search terms: `-a`, `--allow`, `--audit`, `--neverallow`, or `--type`. |
| `-l, --lineno` | In the search results, specify the line number in `policy.conf`. This option is ignored when you search a binary policy. |

**Table 6-1. Options for `sesearch`**

## 6.2. Using seaudit for Audit Log Analysis

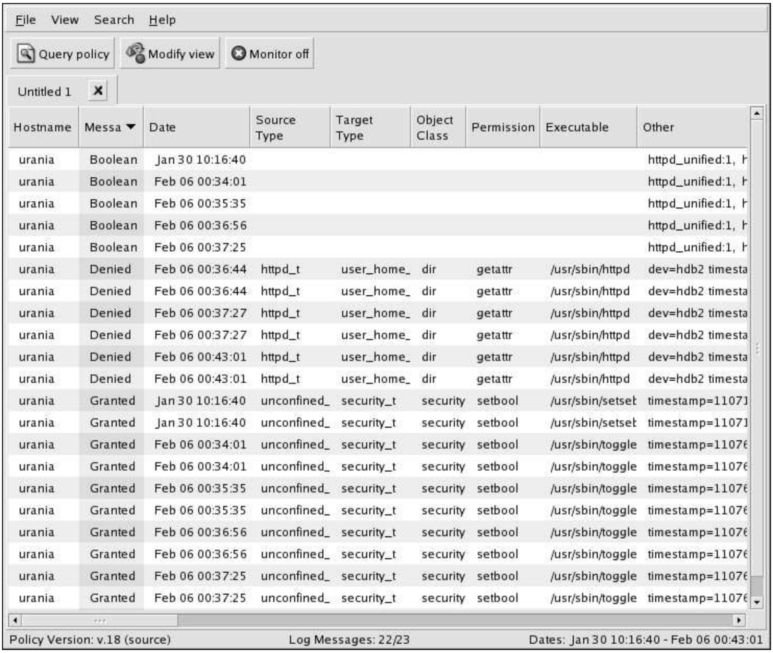
Troubleshooting policy violations can mean wading through convoluted audit logs. The **seaudit** application is designed to help you read, sort, and query your SELinux audit messages. In addition, `seaudit-report` generates formatted reports of SELinux messages from the audit log, useful for reports such as those generated by `logwatch`. The information you gather helps you in analyzing problems and creating solutions.

It is necessary to have super-user privileges to run **seaudit**, because it looks into system logs. For this reason, `/usr/bin/seaudit` is a symlink to **consolehelper**, as well as a program accessible directly by root at `/usr/sbin/seaudit`.

You can choose which log and policy file to use when starting the application, for example, `seaudit -l /path/to/log -p $SELINUX_SRC//policy.conf`. **seaudit** can use both binary and source policy files.

Although simpler than the related **apol**, **seaudit** has more capabilities than are covered by this section. This section focuses on how to accomplish basic tasks using **seaudit**. For more information about what **seaudit** is, read the online documentation at `/usr/share/doc/setools-<`*`version`*`>/seaudit_help.txt`, which is also available from the **Help** menu in **seaudit**.

Figure 6-1 shows **seaudit** displaying the audit log with several different kinds of messages displayed. The **Other** column is where the timestamp and serial number are displayed.



| Hostname | Messa | Date | Source Type | Target Type | Object Class | Permission | Executable | Other |
|---|---|---|---|---|---|---|---|---|
| urania | Boolean | Jan 30 10:16:40 | | | | | | httpd_unified:1, h |
| urania | Boolean | Feb 06 00:34:01 | | | | | | httpd_unified:1, h |
| urania | Boolean | Feb 06 00:35:35 | | | | | | httpd_unified:1, h |
| urania | Boolean | Feb 06 00:36:56 | | | | | | httpd_unified:1, h |
| urania | Boolean | Feb 06 00:37:25 | | | | | | httpd_unified:1, h |
| urania | Denied | Feb 06 00:36:44 | httpd_t | user_home_ | dir | getattr | /usr/sbin/httpd | dev=hdb2 timesta |
| urania | Denied | Feb 06 00:36:44 | httpd_t | user_home_ | dir | getattr | /usr/sbin/httpd | dev=hdb2 timesta |
| urania | Denied | Feb 06 00:37:27 | httpd_t | user_home_ | dir | getattr | /usr/sbin/httpd | dev=hdb2 timesta |
| urania | Denied | Feb 06 00:37:27 | httpd_t | user_home_ | dir | getattr | /usr/sbin/httpd | dev=hdb2 timesta |
| urania | Denied | Feb 06 00:43:01 | httpd_t | user_home_ | dir | getattr | /usr/sbin/httpd | dev=hdb2 timesta |
| urania | Denied | Feb 06 00:43:01 | httpd_t | user_home_ | dir | getattr | /usr/sbin/httpd | dev=hdb2 timesta |
| urania | Granted | Jan 30 10:16:40 | unconfined_ | security_t | security | setbool | /usr/sbin/setset | timestamp=11071 |
| urania | Granted | Jan 30 10:16:40 | unconfined_ | security_t | security | setbool | /usr/sbin/setset | timestamp=11071 |
| urania | Granted | Feb 06 00:34:01 | unconfined_ | security_t | security | setbool | /usr/sbin/toggle | timestamp=1107€ |
| urania | Granted | Feb 06 00:34:01 | unconfined_ | security_t | security | setbool | /usr/sbin/toggle | timestamp=1107€ |
| urania | Granted | Feb 06 00:35:35 | unconfined_ | security_t | security | setbool | /usr/sbin/toggle | timestamp=1107€ |
| urania | Granted | Feb 06 00:35:35 | unconfined_ | security_t | security | setbool | /usr/sbin/toggle | timestamp=1107€ |
| urania | Granted | Feb 06 00:36:56 | unconfined_ | security_t | security | setbool | /usr/sbin/toggle | timestamp=1107€ |
| urania | Granted | Feb 06 00:36:56 | unconfined_ | security_t | security | setbool | /usr/sbin/toggle | timestamp=1107€ |
| urania | Granted | Feb 06 00:37:25 | unconfined_ | security_t | security | setbool | /usr/sbin/toggle | timestamp=1107€ |
| urania | Granted | Feb 06 00:37:25 | unconfined_ | security_t | security | setbool | /usr/sbin/toggle | timestamp=1107€ |

Policy Version: v.18 (source)    Log Messages: 22/23    Dates: Jan 30 10:16:40 - Feb 06 00:43:01

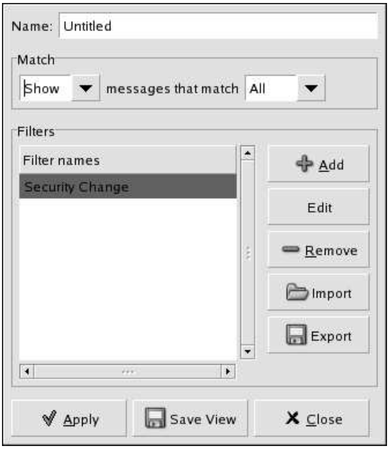
**Figure 6-1. seaudit Showing `$AUDIT_LOG`**

### 6.2.1. Arranging Your Views in seaudit

There are several features to **seaudit** that make it easier to work with the audit messages. The first happens simply by loading a log into **seaudit**. You find only the SELinux log entries are displayed, with all of the data fields in the log message divided into columns. Clicking on the top of a column sorts the records by that column.

If you want real time monitoring of the log file, click on **Monitor** <***off***> to toggle the log watching. Clicking on the button again turns monitoring off.

Column sorting only supports one level, meaning you can only sort by a single column. The **Other** column is not a sort option. In order to sort by more fields, use the filter capability through **View** => **Modify** or the **Modify view** button. The window that pops up manages your filters, letting you control, edit, save (**Export**), and load (**Import**) the filters, as well as save the entire view:



Name: Untitled

Match

Show messages that match All

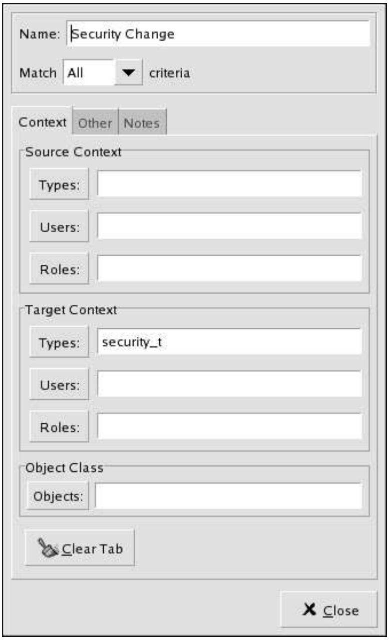
Filters

Filter names

Security Change

**Figure 6-2. seaudit View Filter**

The **View** window is where you create filters to help organize and analyze log entries. Clicking on the **Add** or **Edit** button brings up the **Edit filter** window:



Name: Security Change

Match All criteria

Source Context

Types:

Users:

Roles:

Target Context

Types: security_t

Users:

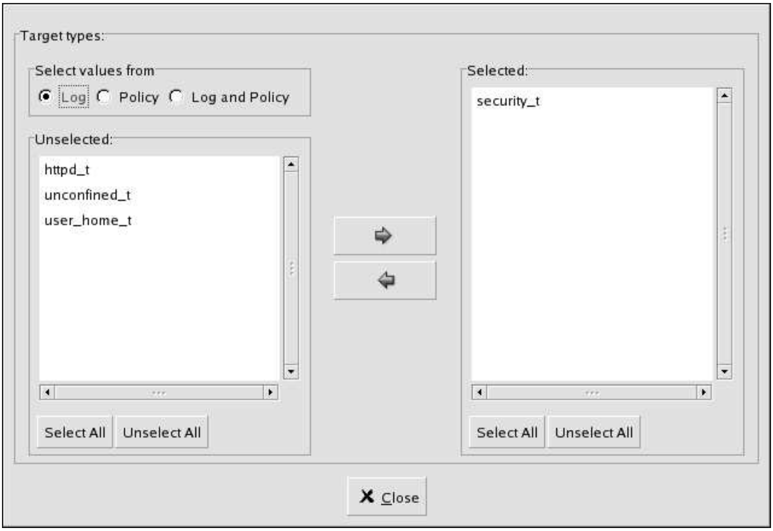
Roles:

Object Class

Objects:

**Figure 6-3. seaudit Edit Filter**

From here you can set filters to sort on source context, target context, object class, IP address, port, interface, source executable, path to the target, and hostname. Criteria matching is all or any. When you choose a context for a filter, such as under **Target Type** clicking on **Types:**, the **Select Target Types** window pops up with the available types, as shown in Figure 6-4. These types are just the ones that are represented in the audit logs:


Target types:

Select values from: ● Log ○ Policy ○ Log and Policy

Unselected:
- httpd_t
- unconfined_t
- user_home_t

Selected:
- security_t

**Figure 6-4. seaudit Editing Filters Select Target Types Window**

Instead of picking exact types, you can use wildcard and related matching mechanisms, collectively called *globbing*. The techniques are detailed in Table 6-2. All globbing expressions are case sensitive.

| Glob Type | Behavior |
|---|---|
| ? | Matches any single character. For example, a?c matches abc, aqc, a1c, and so forth. |
| * | Matches one or more characters. For example, reg* matches regular, regex, regexp, and so forth. |
| [...] | A list of characters searched for in a particular position. For example, a[bde]c matches abc, adc, and aec. The brackets *cannot* be empty. |
| [x-y] | This searches for the range of specified characters. The range is [*<beginning_of_range>*-*<end_of_range>*]. You can specify more than one range. For example, h[a-z]ly matches any word that begins with an h and ends with an ly and has only one alpha character in between, such as hrly, hmly, and hlly. Another example is 1[2-46-9]0, which gives you every combination except 150. The range is inclusive, for example, [a-c] includes a, b, and c, and not just b. |

| Glob Type | Behavior |
|---|---|
| `[!...]` | When the `!` opens the expression, it signifies a complement of the character or range that follows. This globbing pattern uses a list. Complementation is a mathematical term, and in this cases means, "not the character or range that follows." For example, `1[!4-6]0` matches any combination from `110` to `190` except `150`, that is, "not the number that is in the range of 4 to 6." The range is not inclusive, which means `[!4-6]` can only mean `!5`, and does not mean `!4`, `!5`, `!6`. |
| `^ $` | The caret and the dollar sign signify that you want to anchor the search so that the string itself is searched for without anything before or after it. For example, ^httpd_suexec$ is searching for just `httpd_suexec` and not `httpd_suexec_exec_t`. To open up one end of the search, replace the `^` or `$` with a different globbing mechanism. |

**Table 6-2. Globbing Expressions in seaudit**

You can combine the globbing expressions for increased capability. For example, `h[a-z]*ly` finds patterns such as `hzfooly`, `hazily`, `hardly`, `hardily`, and so forth.

All of these search functions are described in additional detail with examples in the **seaudit** help documentation. This is viewable through **Help** => **Help**, which opens the file from `/usr/share/doc/setools-<`*`version`*`>/seaudit_help.txt`.

You can open multiple views into the same log by opening additional tabs. **View** => **New** creates a new view of the log in a different tab, and you can sort and filter this log. This helps when you are sorting through complex denials trying to find root causes.

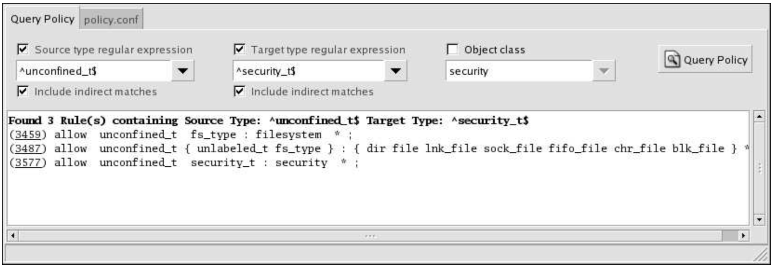
Back in the main audit log view window, you can get more information on each individual log entry. By double-clicking on the entry, or right-clicking and choosing **View Entire Message**, the individual log entry is opened in a pop-up window. You can also get to this view through **View** => **View Entire Message**, which displays the selected message. If more than one message is selected, the top one is displayed.

There are two additional right-click commands. **Query Policy Using Message** opens a **Query Policy** window with the search parameters populated with details from that single message. This is explained in Section 6.2.2 *Searching and Querying in* ***seaudit***. The last right-click menu option is **Export Message to File**, which lets you save a log file containing the single message.

## 6.2.2. Searching and Querying in seaudit

A more complex method of analyzing your audit messages is to look for pertinent rules in the policy. You can use the elements of the denial message in the queries. The query tool in **seaudit** is similar to the TE rules query capability in **apol**. This is helpful for conducting SELinux work involving log and policy analysis.

Clicking on **Query policy** or **Search** => **Query policy** opens the **Query Policy** window. If you have a log highlighted, the fields are pre-filled with the log entry details.



☑ Source type regular expression: ^unconfined_t$
☑ Include indirect matches

☑ Target type regular expression: ^security_t$
☑ Include indirect matches

□ Object class: security

```
Found 3 Rule(s) containing Source Type: ^unconfined_t$ Target Type: ^security_t$
(3459) allow  unconfined_t  fs_type : filesystem  * ;
(3487) allow  unconfined_t { unlabeled_t fs_type } : { dir file lnk_file sock_file fifo_file chr_file blk_file } *
(3577) allow  unconfined_t  security_t : security  * ;
```

**Figure 6-5. seaudit Query policy Window**

Full regular expression support is enabled for the **Query policy** window. The globbing expression behavior used in the **Modify view** filtering is not available.

In the **Query policy** window, the **policy.conf** tab displays the currently active `policy.conf` from the active policy. You need to have a policy file loaded in order to query the policy.

If you do not have the policy source installed or the file `$SELINUX_SRC/policy.conf` is not present, you need to manually load a different policy file through **File => Open policy**. For example, you can use the binary policy in `$SELINUX_POLICY/policy.<XY>`, or a binary or source policy file from another system. However, if you use a binary policy, the **policy.conf** tab does not appear.

With the policy loaded into the **policy.conf** tab, your query results include a number in parenthesis, for example, `(3577)`. These numbers are hyperlinks to the corresponding line number in the `policy.conf` file. Clicking on the hyperlink takes you directly to the location in the **policy.conf** tab.

In the query fields, checking **Include indirect matches** ensures that you are searching by the source and target values as well as any attributes that contain types identified by those same regular expressions. Unchecking a set of fields, such as **Target type regular expression**, disables that set from the query. This opens the query up to finding, for example, every connection from the source to every target. To truly open the search, you can remove the **Object class** query field.

## 6.2.3. Using `seaudit-report` to Generate Reports

The utility `seaudit-report` is useful for generating reports of SELinux-related log activity. The command lets you specify the incoming log source, either from files or STDIN, and output to a file or STDOUT as text or styled HTML. By piping through `seaudit-report` using STDIN and STDOUT, you can use this utility to generate automatic reports that can be sent via email or posted on an Intranet page. The format is designed to be used by programs such as `logwatch`.

You can customize the reports generated by `seaudit-report` in two ways. Visual customization is first done in the layout of the configuration file, which determines which reports are nested where. This continues if you use HTML output. The *cascading stylesheet* (*CSS*) at `/usr/share/setiils/seaudit-report.css` can be used directly or modified to fit your needs.

Another customization is through the `seaudit-report` report configuration file at `/usr/share/setools/seaudit-report.conf`. This file details how to enable and disable the standard report fields, as well as how to include customized views that have been saved from **seaudit**. Here is the default configuration XML:

```
<seaudit-report title="SEAudit Log Report">
        <standard-section id ="Statistics" title="Log \
Statistics"></standard-section>
        <standard-section id ="PolicyLoads" title="Policy \
Loads"></standard-section>
        <standard-section id ="EnforcementToggles" \
title="Enforcement mode toggles"></standard-section>
```

```
        <standard-section id ="PolicyBooleans" title="Policy \
boolean changes"></standard-section>
        <standard-section id ="AllowListing" title="Allow \
Listing"></standard-section>
        <standard-section id ="DenyListing" title="Deny \
Listing"></standard-section>
</seaudit-report>
```

You can remove reports by removing the XML tag-set for it, and you can put custom views obtained from saved views in **seaudit** into a <`custom-section`>. Refer to the installed documentation at `/usr/share/setools/seaudit-report.conf` for a more detailed explanation of usage.

This shows portions of a report output from `seaudit-report`, rendered directly to plain text:

```
# Begin

# Report generated by seaudit-report on
# Sun Feb  6 16:20:01 2005

Title: SEAudit Log Report
Log Statistics
--------------
Number of total messages: 15
Number of policy load messages: 2
Number of policy boolean messages: 2
Number of allow messages: 4
Number of denied messages: 8

Policy Loads
------------
Number of messages: 2

Feb 06 17:32:37 urania kernel: security: 3 users, 4 roles, \
  316 types, 20 bools
Feb 06 17:32:37 urania kernel: security: 53 classes, 9815 rules

Enforcement mode toggles
------------------------
Number of messages: 0


Policy boolean changes
----------------------
Number of messages: 2

Feb 06 19:44:32 urania kernel: security: committed booleans: \
{ httpd_unified:1, httpd_enable_cgi:1, httpd_enable_homedirs:1, \
httpd_ssi_exec:1, httpd_tty_comm:0, httpd_disable_trans:1, \
dhcpd_disable_trans:1, mysqld_disable_trans:0, \
named_disable_trans:0, named_write_master_zones:0, \
nscd_disable_trans:0, ntpd_disable_trans:0, \
portmap_disable_trans:0, postgresql_disable_trans:0, \
snmpd_disable_trans:0, squid_disable_trans:0, \
syslogd_disable_trans:0, winbind_disable_trans:0, \
ypbind_disable_trans:0, allow_ypbind:1 }
...

Allow Listing
-------------
Number of messages: 4

Feb 06 19:44:32 urania kernel: audit(1107747872.351:7619987): \
avc: granted { setbool } for pid=3803 exe=/usr/sbin/togglesebool \
scontext=root:system_r:unconfined_t \
tcontext=system_u:object_r:security_t tclass=security
...

Deny Listing
------------
Number of messages: 8

Feb 06 19:42:45 urania kernel: audit(1107747765.871:7550947): \
avc: denied { getattr } for pid=2479 exe=/usr/sbin/httpd \
path=/home/auser/public_html dev=hdb2 ino=921135 \
scontext=user_u:system_r:httpd_t \
tcontext=system_u:object_r:user_home_t tclass=dir

Feb 06 19:42:45 urania kernel: audit(1107747765.872:7550962): \
avc: denied { getattr } for pid=2479 exe=/usr/sbin/httpd \
path=/home/auser/public_html dev=hdb2 ino=921135 \
scontext=user_u:system_r:httpd_t \
tcontext=system_u:object_r:user_home_t tclass=dir
...

# End
```

## 6.3. Using apol for Policy Analysis

There are many aspects to a formal security policy analysis. In this guide, *policy analysis* refers to analyzing SELinux policy to discover the relationship between types defined in the policy. This section presents **apol**, which is designed specifically for analyzing policy.

Policy analysis is not only performed on running systems, but is an integral part of designing and writing a policy. You can analyze your custom policy using **apol** as part of your testing process, before you load it on a machine. **apol** can help you discover unexpected and undesirable results of your policy writing decisions. It helps show the differences between versions or kinds of policy. For example, you can analyze each iteration of your policy, reusing saved queries, looking for information leaks or unwanted transitions.

Policy analysis with **apol** is many magnitudes more complex than audit log analysis with **seaudit**. The `setools` package comes with several important documents to read in order to understand how to properly utilize **apol**, as well as how to interpret your results. Reading the documentation from `/usr/share/doc/setools-<version>` is recommended:

- `apol_help.txt` — This detailed help file describes how to use all of the features of **apol**, as well as a walkthrough of the tabbed interface.
- `dta_help.txt` — This is an overview of *domain transition analysis* (*DTA*), which studies the ability of processes to change their domains in a particular policy.
- `iflow_help.txt` — This is an overview of *information flow analysis*, which finds the expected and unexpected possible routes information can travel between two types in a policy.
- `types_relation_help.txt` — This help file discusses analyzing the relationship between two types. Information flow analysis is essentially what you are doing when you analyze the policy.

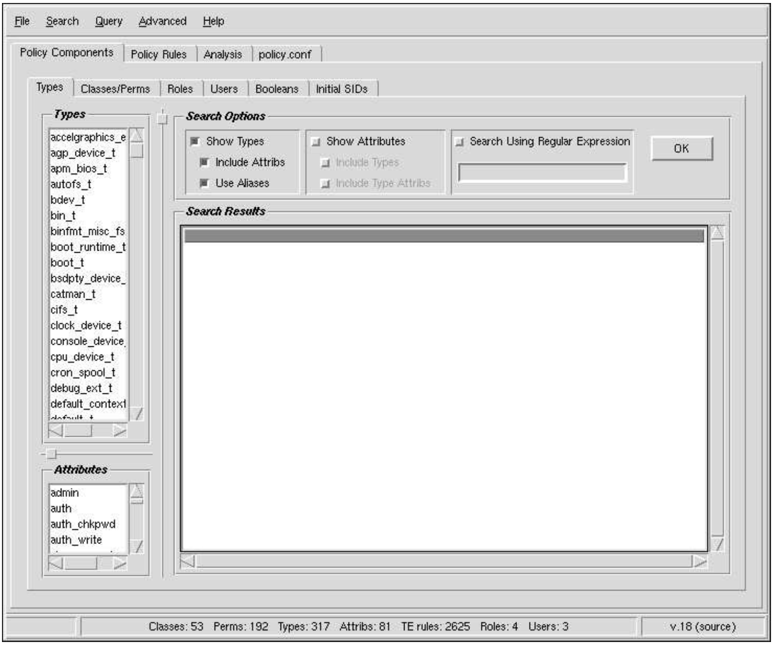
The topics each of these help files covers is central to the task of analyzing SELinux policy. The **apol** UI is organized around these tasks. The following sections explain these tasks, discussing how to utilize the GUI for your analysis work.

**Note**

> Both the source `policy.conf` and binary `policy.<XY>` files can be analyzed by **apol**. Much of the results are similar, but there are noteworthy differences. This is because the binary compilation process strips out attributes as well as the initial SIDs. It is the lack of attributes that most affects the analysis process. When analyzing a binary policy, attributes cannot be included as search parameters.
>
> The **policy.conf** tab is disabled for the binary policy, as well as the **Initial SIDs** tab under the **Policy Components** tab. The field **Attributes** is empty, and although you can select **Attrib(ute)s** in various search parameters, it has no effect when analyzing a binary policy.

## 6.3.1. Policy Component Analysis

When opening the policy file, **apol** gathers and organizes information. The same information is difficult to identify and extrapolate manually going through the policy files. For example, there is no master list within the policy source of which types belong to which attributes. This information is scattered throughout the policy. **apol** gathers and displays these SELinux categories.



Types:
accelgraphics_e
agp_device_t
apm_bios_t
autofs_t
bdev_t
bin_t
binfmt_misc_fs
boot_runtime_t
boot_t
bsdpty_device_
catman_t
cifs_t
clock_device_t
console_device
cpu_device_t
cron_spool_t
debug_ext_t
default_context

Search Options:
☒ Show Types
☒ Include Attribs
☒ Use Aliases
□ Show Attributes
□ Include Types
□ Include Type Attribs
□ Search Using Regular Expression

Attributes:
admin
auth
auth_chkpwd
auth_write

Classes: 53 Perms: 192 Types: 317 Attribs: 81 TE rules: 2625 Roles: 4 Users: 3 v.18 (source)

**Figure 6-6. apol with `policy.conf` Loaded**

Figure 6-6 shows the **Policy Components** tab. Within this tab there are tabs for **Types**, **Classes/Perms**, **Roles**, **Users**, **Booleans**, and **Initial SIDs**. Under each tab is the capability to perform basic searches.

**Note**

There are declared types that do not have any rules written for them or file contexts set for them. For example, `swapfile_t` is declared in `$SELINUX_SRC/types/file.te`, so it appears in the **Types** menu within the **Types** tab. However, the file type is not assigned to any file nor are there rules about it.

If you are wondering if a particular type is used in the policy, you can search for it under the **Policy Rules** tab. If no rules are found, then it is an unused type.

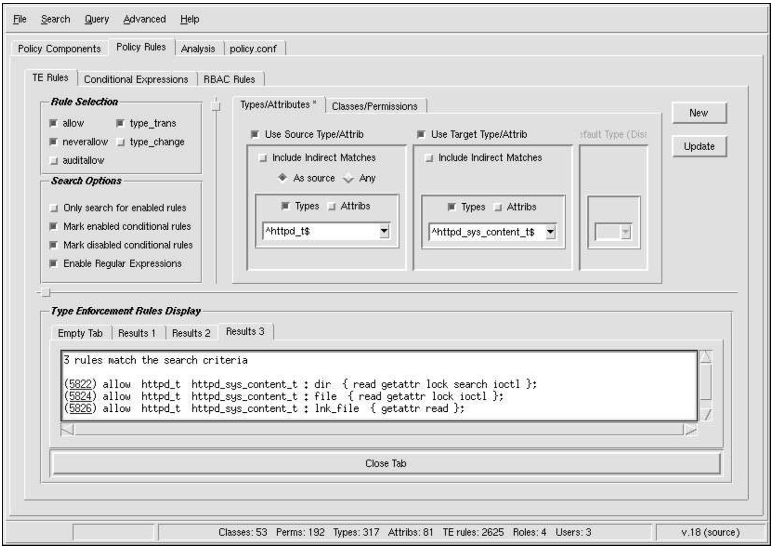
**Tip**

One feature of the **Booleans** tab is that you can set Boolean values within the policy loaded into **apol**. This does not affect the Boolean value on the disk or in memory. This lets you test the effect on the policy of changing different Booleans, entirely within **apol**. You can then do TE rule and information flow analysis with the new Boolean settings.

## 6.3.2. TE Rule Analysis

Rule analysis looks into the relationship between a pair of types, trying to find the ways they interact. The interaction could be direct or indirect due to the use of attributes, and enabled or disabled by a Boolean setting.

Under the **Policy Rules** tab are search options and regular expression fields for defining the source and target type or attribute. The **Rule Selection** menu lets you choose the kind of rule, such as `allow`, `neverallow`, and `auditallow`. In Figure 6-7, the menu for **Default Type** is squeezed in the image since it is disabled:



Rule Selection

☒ allow ☒ type_trans
☒ neverallow □ type_change
□ auditallow

Search Options

□ Only search for enabled rules
☒ Mark enabled conditional rules
☒ Mark disabled conditional rules
☒ Enable Regular Expressions

☒ Use Source Type/Attrib
□ Include Indirect Matches
● As source ○ Any
☒ Types □ Attribs
^httpd_t$

☒ Use Target Type/Attrib
□ Include Indirect Matches
☒ Types □ Attribs
^httpd_sys_content_t$

Type Enforcement Rules Display

```
3 rules match the search criteria

(5822) allow  httpd_t  httpd_sys_content_t : dir  { read getattr lock search ioctl };
(5824) allow  httpd_t  httpd_sys_content_t : file  { read getattr lock ioctl };
(5826) allow  httpd_t  httpd_sys_content_t : lnk_file  { getattr read };
```

Classes: 53 Perms: 192 Types: 317 Attribs: 81 TE rules: 2625 Roles: 4 Users: 3 v.18 (source)

**Figure 6-7. Policy Rules and TE Rules Search**

The **Search Options** menu lets you pick search parameters. The selection for **Only search for enabled rules** refers to the Boolean value for a rule, or if a conditional expression (`ifdef` statement) is true. This selection is a filter that can hide or reveal a large number of possible routes between a pair of types.

If you expand your search to include disabled conditional rules, you can have the results highlighted. By selecting **Mark enabled conditional rules** and **Mark disabled conditional rules**, the conditional rules are identified and marked with their status of **Enabled** or **Disabled**.

The search parameter tabs **Types/Attributes** and **Classes/Permissions** let you describe details about the source and target you are analyzing. An asterisk * appears on the tab if a parameter from that tab is set and affecting the search. You can define the source by using the exact type or using a regular expression. Automatically, the expression is anchored at both ends, so a caret `^` and dollar sign `$` surround the search terms unless you explicitly change them.

You can choose to **Include Indirect Matches**, which expands the search to include attributes. You may choose to search by type and/or by attribute, with searching by attribute being similar to including indirect matches.

You can further refine or expand the search using the object classes and associated permissions under the **Classes/Permissions** tab.

The search results are displayed in a tabular format, with a different search result and its search parameters for each tab. Switching between search result tabs changes the search parameters. You can keep up to ten results open at a time.

To start a new search, you click **New**, which displays the results in a new tab. You can change the search parameters and click **Update**, which updates the search within the existing tab. This allows you to keep track of many different parts of an analysis. You can save queries for later recall using **Query => Save Query**.

Example 6-1 shows the contents of a search result field from the **Type Enforcement Rules Display** area. The search that generated these results included searching for enabled and disabled conditional rules with display. The type searched for is `^httpd_t$` as a source and/or target. The comments following the mark ## are explanations inserted for this guide and are not part of the standard **apol** output.

```
278 rules match the search criteria
Number of enabled conditional rules: 23
Number of disabled conditional rules: 34

(3813) allow httpd_t var_log_t:dir { read getattr lock \
  search ioctl add_name write };
(3815) allow httpd_t httpd_log_t:file { create ioctl read \
  getattr lock append };
(3821) allow httpd_t httpd_log_t:dir { setattr read \
 getattr lock search ioctl add_name write };
(3825) allow httpd_t httpd_log_t:lnk_file read;
(3882) allow httpd_t unconfined_t:fd use;
(3884) allow httpd_t unconfined_t:process sigchld;

## These are related to the Boolean httpd_disable_trans,
## showing that it is not set to true:

(4024) allow unconfined_t httpd_t:process transition; [Enabled]
(4074) allow httpd_t unconfined_t:process sigchld; [Enabled]
(4086) allow httpd_t unconfined_t:fd use; [Enabled]
(4088) allow unconfined_t httpd_t:fd use; [Enabled]
(4098) allow httpd_t unconfined_t:fifo_file { ioctl read \
  getattr lock write append }; [Enabled]
(4108) allow httpd_t httpd_exec_t:file { read getattr lock \
  execute ioctl }; [Enabled]
(4118) allow httpd_t httpd_exec_t:file entrypoint; [Enabled]
(4126) allow unconfined_t httpd_t:process { noatsecure \
  siginh rlimitinh }; [Enabled]

## These are part of other httpd_* Booleans that are set
## to false in the file /etc/selinux/targeted/booleans:

(4554) allow httpd_t httpd_sys_script_t:process transition; \
  [Disabled]
(4594) allow httpd_t httpd_sys_script_exec_t:file { read getattr \
  execute }; [Disabled]
(4604) allow httpd_sys_script_t httpd_t:process sigchld; [Disabled]
(4616) allow httpd_sys_script_t httpd_t:fd use; [Disabled]
(4618) allow httpd_t httpd_sys_script_t:fd use; [Disabled]
```

**Example 6-1. apol TE Rules Search Results**

Within the search results, there are hyperlinks to the left of each rule. The number corresponds to the line number in `policy.conf`, and clicking on, for example, **(3813)** switches your view to the **policy.conf** tab, taking you directly to line 3813. These hyperlinks are only visible if you have **apol** analyzing the `policy.conf` file.

If you are using a binary policy file such as `policy.18`, the rules are compiled and not available for viewing. The top-level tab **policy.conf** is not present when analyzing the binary policy.

There are two other search capabilities within the **Policy Rules** tab, the **Conditional Expressions** and **RBAC Rules** tabs.

The **Conditional Expressions** tab allows you to search just the conditional expressions, viewing the rules within them. The only searchable rule types are `allow`, `audit`, and `transition`. All conditional expressions are displayed in the default view; you can narrow the view using **Search Options**.

You can search either by specific Boolean or with regular expressions. You can reduce the quantity of output by deselecting **Display rules within conditional expression(s)**.

Each rule is marked if it is [enabled] or [disabled], which shows the current state of that conditional in the policy. Changing a value in the **Booleans** tab within the **Policy Components** tab is reflected in the **Conditional Expressions Display** by running your search again. This is another way of analyzing the consequences of toggling a Boolean value, entirely within **apol**.

The last tab under the **Policy Rules** tab is the **RBAC Rules** tab. Effectively, you only use the **Allow** choice, since role transitions are deprecated. Using the **Transition** selection is only useful in analyzing roles in older versions of the SELinux policy, which will not appear in Red Hat Enterprise Linux. The **Default Role** menu is disabled because it is only used in the deprecated role_transition analysis.

To search, set a **Source Role**, either as source or any if you want to search for it in both positions in an allow rule. Then you set a **Target Role** value. The search result shows if the source role may assume the targeted role.

## 6.3.3. Domain Transition Analysis

Domain transition is one important aspect of TE. Since being in a particular domain is the key to controlling that domain's derivative types, the strict control of what domains can transition to what other domains is essential to SELinux security.

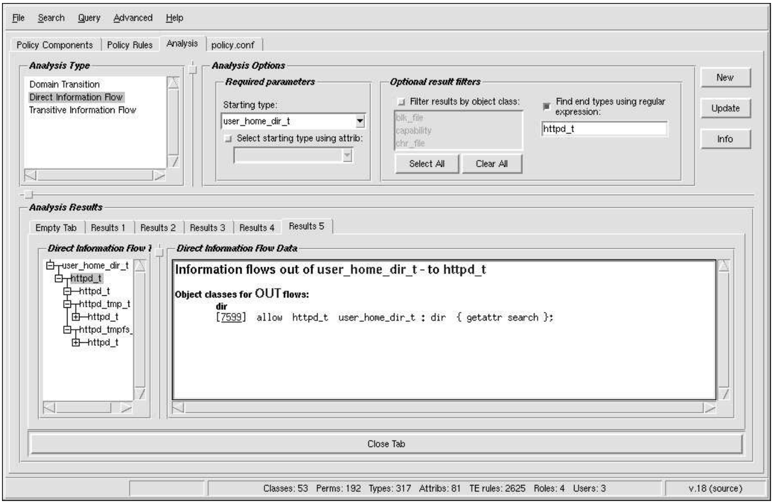
Domain transition is looked at from two directions, forward and reverse. In a forward analysis, you select a source type and search to find all target types it can transition to. You can use search parameters to refine the results. For a reverse analysis, you select a target type and discover all the source types that can transition to the target type.

For a domain transition to occur, there must be three particular allow rules. These rules control the source domain that is attempting to transition to the target domain. One rule permits the process transition itself, a second rule allows the source access to the entry point executable for the target domain, and the third rule allows the target domain itself to use the executable as an entry point.

Domain transition analysis with **apol** centers around identifying these three rules. In some cases, more than one appropriate rule is found. The extensive help file, /usr/share/doc/setools-<*version*>/dta_help.txt, is useful in explaining this analysis.

## 6.3.4. Direct and Transitive Information Flow

Information flow analysis is a central and challenging part of analyzing an SELinux policy. Your analysis may find unexpected or dangerous information flows. For example, if you want to be sure that the content in your /home/ directories (user_home_dir_t) is flowing as you configured it into the httpd_t domain, **apol** searches through the policy to reveal all the ways information flows between the two types. This search is show in Figure 6-8:



Starting type: user_home_dir_t

□ Select starting type using attrib:

□ Filter results by object class:

■ Find end types using regular expression: httpd_t

```
user_home_dir_t
  httpd_t
    httpd_t
    httpd_tmp_t
      httpd_t
    httpd_tmpfs_
      httpd_t
```

Information flows out of user_home_dir_t - to httpd_t

Object classes for OUT flows:

```
        dir
        [7599]  allow  httpd_t  user_home_dir_t : dir  { getattr search };
```

Classes: 53 Perms: 192 Types: 317 Attribs: 81 TE rules: 2625 Roles: 4 Users: 3

v.18 (source)

**Figure 6-8. Direct Information Flow Analysis**

Information flow analysis can be a challenging and daunting task. The policy holds thousands or tens of thousands of rules with hundreds of types, all interacting in multiple ways. The help file `/usr/share/doc/setooles-<version>/iflow_help.txt` is essential reading for understanding information flow analysis in SELinux.

In doing transitive information flow analysis, **apol** attempts to string together different direct flows, looking for ways that information can transit between direct flows. This looks for ways that allow the farthest ends of the different direct flows to pass information to each other.

## 6.4. Performance Tuning

The major performance hit that SELinux can make on the system is in the kernel, where the hooks used through LSM divert the kernel flow into the AVC. Usually, the working set of cached permissions used in normal system operations is relatively small, fewer than 100 AVC entries for most systems with a focused mission. SELinux maintains up to 512 entries in the cache, and does not usually need to perform additional lookups outside of that cache.

If you suspect you are having performance problems due to SELinux or you generally want to fine tune your system, you can monitor the AVC through the `/selinux` file system. The first file, `/selinux/avc/hash_stats`, shows the number of entries, the number of hash buckets used by the entries, and the length of the longest hash chain:

```
cat /selinux/avc/hash_stats
entries: 521                # total number of AVC entries
buckets used: 285/512       # total number of buckets
longest chain: 6            # hash chain of less than 10 is
                            # optimal
```

If your hash chains are growing to be larger than 10, there may be a performance impact. You can consider reducing the size of the cache. To increase or decrease the size of the cache, you can set a new value through this tunable:

```
cat /selinux/avc/cache_threshold
512
echo 768 > /selinux/avc/cache_threshold

# Check to be sure the change took hold.  Be sure you are
# root when using the targeted policy.

cat /selinux/avc/cache_threshold
768
```

**Caution**

The default value of 512 for the cache threshold in Red Hat Enterprise Linux is set from extensive optimization benchmarking. Changing this value could have negative effects on system performance.

To be sure adjusting the cache limit is having positive effects on your performance, watch the number of reclaimed cache entries. Stale cache entries can build up following boot or long after daemon startup, which requires reclaiming entries when more are required for new processes. If you have a system where there are a high number of entries changing across a broad enough policy, this reclamation may occur more often and effect system performance. You can watch the `reclaims` column in the output of `avcstat` using the `-c` option, which displays the cumulative values:

```
avcstat -c 1
... reclaims ...
...      800 ...
...      830 ...
...      876 ...
...      912 ...
...      955 ...
...      992 ...
```

Occasional reclaim activity is within the bounds of normal, and it may increase when changing workloads. Excessive reclaims over a sustained period of time should be looked into.

# Chapter 7.

# Compiling SELinux Policy

**Warning**

> The commands and steps covered in this chapter may render your system inoperable or unable to be supported.
>
> Nothing in this chapter should be performed on a production system without having been thoroughly tested in a development or sandbox environment first.
>
> If you are going to compile and install a custom policy, be prepared to take the actions you need to safeguard your data and installation. Proper backup procedures, change-reversal plans, and an informed methodology are key to your success.

This chapter covers the considerations and methods for compiling SELinux policy. Following instructions on compiling SELinux policy, this chapter presents some reference information and considerations.

## 7.1. Policy Compile Procedure

Policy is usually compiled to enable a customization to take effect on your system. You may also compile policy under development, such as when working on writing a new policy or SELinux-aware application.

When you install a new policy, you must eventually reboot to test that it works during system start-up. If the policy change is significant enough, such as installing an entirely new policy, you need to reboot to ensure all applications are running in the right context for the loaded policy. This is similar to any major configuration change under Linux; you want to be sure it works properly from system start-up on at least one production-equivalent machine.

**Note**

> Policy updates from Red Hat should not require a reboot after installation. If a reboot were required, that fact would be clearly noted in the package advisory.
>
> A reboot is required when the policy change is significantly different. For example, switching from the targeted to the strict policy requires a reboot. Ordinary policy updates do not.

To compile SELinux policy:

**Compiling the SELinux Policy**

1. `cd /etc/selinux/targeted/src/policy/`

2. Policy compiles if there are new or changed files in certain locations in the source tree. Those files must have a later timestamp than `policy.conf`. If you want to compile the policy but cannot because of the timestamp, you can force a compile.

    - To compile the policy, run `make <make_policy_target>`.
    - If you need to force a policy build, run `make -W <users> <load>`. The target can be any from the `Makefile`.

      The `-W` option tells `make` to act as if the command `touch` had been run on the file `users`. This virtual update of the timestamp is only from the perspective of `make`. It triggers the `Makefile` to build and load the policy because the virtual change to the `users` file gives it a later timestamp than the `policy.conf` file.

In order to compile the policy, you need to install the policy source package, `selinux-policy-targeted-sources-<version>`.

The *Significant Policy `make` Targets* list that follows discusses important `make` targets in the SELinux policy sources. The `Makefile` itself has more options that you can explore yourself.

## Significant Policy `make` Targets

`load`

Compiles, installs, and fully loads the policy into memory.

This runs `load_policy` and installs `$SELINUX_POLICY/policy.<XY>` and `/etc/selinux/targeted/contexts/files/file_contexts`. When the policy gets loaded, the file `$SELINUX_SRC/tmp/load` is created. The `Makefile` does not compile the policy as long as nothing has changed in the policy source tree since the creation time on the file `policy.conf`.

`reload`

Compiles, installs, and loads or reloads the policy. Reloading lets you load the policy in runtime even if the file `$SELINUX_SRC/tmp/load` is present and newer than the last changes in policy source.

`policy`

Only compiles the policy, putting the resulting binary policy file into `$SELINUX_SRC/policy.<XY>`. It also creates a new `policy.conf` file, `file_contexts/file_contexts`, and so forth. This is useful for developing policy that you intend to deploy on another machine.

`relabel`

Relabels the file system using the policy sources, based on the file `$SELINUX_SRC/file_contexts/file_contexts`. As explained in Section 5.2.2 *Relabel a File System*, this is *not* the recommended method for relabeling a file system.

`enableaudit`

Enables auditing on all of the policy rules that are marked `dontaudit`. The `enableaudit` target changes the `dontaudit` rules in `policy.conf`, which is then loaded.

```
cd $SELINUX_SRC/
make enableaudit
make load
```

This is useful for troubleshooting if you are getting SELinux denials that are not generating audit messages. This is usually discovered by testing with `setenforce 0` to see if the operation is then allowed.

When enabled, the number of denial messages may be very large. You return to a `dontaudit`-state by running `make clean` and then `make load` in `$SELINUX_SRC/`.

**Tip**

The `Makefile` makes some decisions based on the timestamps of the two policy files `$SELINUX_SRC/policy.conf`, `$SELINUX_SRC/policy.<XY>`, and the file `$SELINUX_SRC/tmp/load/`. Because of this you may run into tricky behavior.

The `Makefile` will only rebuild `policy.conf` if there are newer policy files in the source tree. Two behaviors arise from this:

1. If you move a TE file into the source tree that has a timestamp older than the `policy.conf`, the `Makefile` does not rebuild `policy.conf`.
2. If you move a TE file so that it is no longer in the build path, in particular `$SELINUX_SRC/domains/misc/`, the `Makefile` does not necessarily recognize the change.

When this occurs, `make load` will validate the policy and `touch tmp/load`, but not compile the policy. You can force the compile with `make -W users load`.

The `make load` command compares the timestamp on `tmp/load` and the binary policy file in `$SELINUX_SRC/`. This file, `policy.<XY>`, is created in the policy source directory by the command `make policy`. If the binary policy file is newer than the file `tmp/load`, the policy is loaded.

For example, you write a `local.te` custom policy file, run tests on it, then remove the file from the source tree. With no new files in the policy source tree, the `Makefile` does not reload the policy. This is because the `Makefile` is only looking for new files in the source tree. You must run `make -W users load`, which compiles, installs, and loads the policy. This returns you to an uncustomized state.

Later you decide to use the custom policy again. You retrieve the file from the unused policy source location, moving it back to where it will compile into the policy. However, running `make load` or `make reload` has no effect. This is because the file `local.te` is not a new file, it continues to have the original creation timestamp on it. In order to keep the accurate timestamp on the file, rather than using `touch` you can again use `make -W domains/misc/local.te load`.

## 7.2. What Happens During Policy Build

There are multiple events during policy build, some depending on which `make` target you choose. The end result is a binary policy file, with several ancillary files created in the process, including `policy.conf`. The compilation itself follows the same essential steps regardless of the `make` target:

1. All of the configuration files from the `$SELINUX_SRC/` tree that are used in the policy are concatenated together. This is a pre-processed state.

   The source configuration files are discussed extensively in Chapter 2 *SELinux Policy Overview*. The basic qualification for inclusion is to have a TE file in `$SELINUX_SRC/domains/`, but not in the `domains/unused/` directory.

2. The `m4` pre-processor takes the aggregate configuration input and expands the macros, making the `policy.conf` file.
3. The `checkpolicy` policy compiler runs against `policy.conf`, resulting in the `policy.<XY>` binary policy file being created. This file is installed into `$SELINUX_POLICY/`, where it will be picked up on next system boot. Some `make` targets load the policy into memory during runtime. The `make policy` command builds the policy and puts the binary policy file in the source directory, `$SELINUX_SRC/policy.<XY>`.

During the compilation, several files and a directory are created or updated. The most important is `$SELINUX_SRC/policy.conf`. Also in the `$SELINUX_SRC/` directory is `tmp/`, which contains temporary build files, including `load`. This file is a zero-byte file that is used by the `Makefile` to determine the time the policy was last loaded. Finally, the file `$SELINUX_SRC/file_contexts/file_contexts` is created, which is a concatenation of all of the various file contexts files in the source tree.

At the heart of the compilation is `checkpolicy`. This tool compiles the policy into its binary form, and can also be used to validate the policy. Policy compilation is best left to the `Makefile` to handle, but you can gain some insight into any binary policy file using `checkpolicy`:

```
# By itself, checkpolicy looks for a policy.conf file in the
# current working directory, which might normally be $SELINUX_SRC/.

cd $SELINUX_SRC/
checkpolicy
checkpolicy:  loading policy configuration from policy.conf
security:  3 users, 4 roles, 316 types, 20 bools
security:  53 classes, 9815 rules
checkpolicy:  policy configuration loaded

# You can specify a binary policy file with -b:

checkpolicy -b $SELINUX_POLICY/policy.18
checkpolicy:  loading policy configuration from \
  /etc/selinux/targeted/policy/policy.18
security:  3 users, 4 roles, 316 types, 20 bools
security:  53 classes, 9817 rules
checkpolicy:  policy configuration loaded
```

# Chapter 8.

# Customizing and Writing Policy

**Warning**

The commands and steps covered in this chapter may render your system inoperable or unable to be supported.

Nothing in this chapter should be performed on a production system without having been thoroughly tested in a development or sandbox environment first.

If you are going to compile and install a custom policy, be prepared to take the actions you need to safeguard your data and installation. Proper backup procedures, change reversal plans, and an informed methodology are key to your success.

This chapter discusses troubleshooting and customizing your SELinux policy and presents a methodology for writing policy. Specific cautions are discussed.

**Note**

Presenting a comprehensive guide to writing policy is not within the scope for this book. For more information on writing policy, refer to the resources in Chapter 9 *References*.

For this reason, the policy writing guidelines presented here are generic. Generic ideas are easier to apply to your unique environment.

If the resources and general methodologies are not sufficient for your policy writing needs, contact Red Hat support or sales for information about policy writing services.

## 8.1. General Policy Troubleshooting Guidelines

When troubleshooting, use the kernel boot parameter *`selinux=0`* as a last resort. If using `setenforce` during runtime is not sufficient, try booting with *`enforcing=0`* to switch to permissive mode. You still have SELinux checking enabled and `avc: denied` messages logged to `$AUDIT_LOG`, but the enforcing is disabled.

By troubleshooting with SELinux enabled, you can more easily identify and resolve problems. For example, if SELinux is fully disabled, the `-Z` option is not available for finding the security context of objects. You are not able to relabel a file or the file system with SELinux disabled. Finally, any new files or directories you create have no SELinux security attributes, causing more problems when you boot into SELinux.

Save *`selinux=0`* and `SELINUX=disabled` in `/etc/sysconfig/selinux/` for longer-term disabling.

## 8.2. Minor Customizations of the Existing Policy

You may find it useful to resolve SELinux denials by using new policy rules to allow the behavior. The ramifications on security are impossible to predict. At the worst, you are back to standard Linux security. Your policy changes may be enough to effectively disable the confinement for one or more parts of a targeted daemon policy.

You must take these considerations into account when adjusting the policy. You can use **apol** to analyze the transitions and TE rules, looking for where your policy changes weaken security. For more information about working with **apol**, read Section 6.3 *Using **apol** for Policy Analysis*.

This procedure takes you through using `audit2allow` to add minor custom rules to the existing policy. It assumes that you have one or more `avc: denied` messages in `$AUDIT_LOG`.

**Tip**

Put local policy changes in `$SELINUX_SRC/domains/misc/local.te`, and file context information in `$SELINUX_SRC/file_contexts/misc/local.fc`. Your files are picked up on compile.

Separating your local customizations from the main source is akin to maintaining a set of patches to pristine source code. When you update the policy source, your changes won't be clobbered or at risk.

If you package your own policy, follow a similar methodology. Use the maintained, upstream source for either the supported targeted or the unsupported strict policy. You can rebuild the policy packages, having your changes be in standalone files such as `local.te` or new files within `$SELINUX_SRC/domains/program/`.

If you change the existing TE files, you have to merge the patches with each new policy. Remember that `policy.conf` is all of the various source files concatenated together. It does not need to know where the content came from, that is a decision made in the `Makefile`. The files and directories in $SELINUX_SRC/ are a convenience for the policy writer. Following the methods prescribed by the SELinux developers helps you in your policy writing efforts.

**Warning**

Just because `audit2allow` generates a rule does not make it a good or secure rule.

You must look through the rules generated by `audit2allow` for a sanity check. For example, an application generates an SELinux denial asking for read *and* write permission to `/etc/passwd`. A rule generated by `audit2allow` out of the audit log would allow that permission. You need to understand the underlying application and decide if it *should* have those permissions.

This is one of the ways SELinux can help find flaws in applications, where excessive and unnecessary levels of access are attempted.

### Adding Rules to the Policy Using `audit2allow`

1. Be sure you are in the policy source directory:
   ```
   cd $SELINUX_SRC/
   ```
2. If you have an existing file at `$SELINUX_SRC/domains/misc/local.te`, make a backup before proceeding with the rule creation:
   ```
   # The directory domains/unused is ignored during the
   # policy build and is a safe place for archiving.

   cp domains/misc/local.te domains/unused/local.te.backup
   ```

3. Tell `audit2allow` to look in `dmesg` for denial messages, only since the last `load_policy` ran, and write that to `domains/misc/local.te`:
   ```
   audit2allow -d -l -o domains/misc/local.te
   ```
   Look in `local.te` to be sure you don't have any duplicate rules. This is one reason for having `audit2allow` generate rules since the last `load_policy`, to keep from creating duplicates.

   Once you have a complete set of working rules, you may want to look for ways to rewrite and simplify the rules. One way to do this is to have `audit2allow` run one final time against the entire set of denial messages. It looks for ways to consolidate rules into single lines:
   ```
   # For example, two passes of audit2allow yield these rules:

   allow httpd_t user_home_t:dir getattr;
   allow httpd_t user_home_t:dir search;

   # Looking at all of denials in $AUDIT_LOG
   # may reveal some consolidation of rules, for example:

   audit2allow -d -o domains/misc/consolidated_local.te
   grep "user_home_t:dir" domains/misc/consolidated_local.te
   allow httpd_t user_home_t:dir { getattr search };
   ```
4. Test your policy. Run `make load` and try your previously denied operation(s).
5. At this point you may need to do multiple iterations of these steps. Each additional rule allows the operation to get one step further. You use the subsequent denial to write the next rule, and the process continues.

After multiple iterations, you have a set of rules. Now you want to analyze the rules to be sure they follow the principle of least privilege. This is where policy analysis with **apol** is useful, as described in Section 6.3 *Using **apol** for Policy Analysis*.

To make your policy more elegant and efficient, look for macros that provide you the permissions created by your new rules. You can then scrap one or more rules in favor of a macro, which simplifies code reuse. Ideally, your rules benefit from bug fixes and enhancements to the entire policy because your rules build on the policy as a privately maintained set of rules, relying upon the overall structure of the parent policy.

For example, you are running the targeted policy on a newly installed server that is using your in-house, custom-configured `syslog-ng` instead of `sysklogd`. You find that the policy for `syslogd` does not cover all of the non-standard logging operations that you perform. Some of the additional operational requirements for your `syslog-ng` implementation are to open non-standard files and UDP and TCP ports, as well as call non-standard routines.

The following are a sampling of the `avc: denied` messages you have received:

```
Jan 10 04:02:17 example kernel: audit(1009218137.102:0): \
  avc:  denied  { write } for  pid=6109 exe=/sbin/syslog-ng \
  name=kmsg dev=proc ino=-268435446 \
  scontext=system_u:system_r:syslogd_t \
  tcontext=system_u:object_r:proc_kmsg_t tclass=file
Jan 10 04:02:17 example kernel: audit(1009218137.105:0): \
  avc:  denied  { read } for  pid=16202 exe=/bin/bash name=mtab \
  dev=dm-0 ino=7146016 scontext=system_u:system_r:syslogd_t \
  tcontext=system_u:object_r:etc_runtime_t tclass=file
...
Jan 10 16:20:35 example kernel: audit(1009284205.210:0): \
  avc:  denied  { chown } for  pid=6109 exe=/sbin/syslog-ng \
  capability=0 scontext=system_u:system_r:syslogd_t \
  tcontext=system_u:system_r:syslogd_t tclass=capability
Jan 10 16:20:35 example kernel: audit(1009284205.210:0): \
  avc:  denied  { fowner } for  pid=6109 exe=/sbin/syslog-ng \
  capability=3 scontext=system_u:system_r:syslogd_t \
```

```
  tcontext=system_u:system_r:syslogd_t tclass=capability
Jan 10 16:20:35 example kernel: audit(1009284205.210:0): \
  avc:  denied  { fsetid } for  pid=6109 exe=/sbin/syslog-ng \
  capability=4 scontext=system_u:system_r:syslogd_t \
  tcontext=system_u:system_r:syslogd_t tclass=capability
...
Jan 10 16:20:35 example kernel: audit(1009284205.422:0): \
  avc:  denied  { search } for  pid=1411 exe=/bin/bash \
  name=sbin dev=dm-0 ino=7356417 \
  scontext=system_u:system_r:syslogd_t \
  tcontext=system_u:object_r:sbin_t tclass=dir
Jan 10 16:20:35 example kernel: audit(1009284205.422:0): \
  avc:  denied  { getattr } for  pid=1411 exe=/bin/bash \
  path=/bin/bash dev=dm-0 ino=1245248 \
  scontext=system_u:system_r:syslogd_t \
  tcontext=system_u:object_r:shell_exec_t tclass=file
Jan 10 16:20:35 example kernel: audit(1009284205.423:0): \
  avc:  denied  { getattr } for  pid=1411 exe=/bin/bash \
  path=/bin/rm dev=dm-0 ino=1245243 \
  scontext=system_u:system_r:syslogd_t \
  tcontext=system_u:object_r:bin_t tclass=file
Jan 10 16:20:35 example kernel: audit(1009284205.423:0): \
  avc:  denied  { execute_no_trans } for  pid=1411 \
  exe=/bin/bash path=/bin/rm dev=dm-0 ino=1245243 \
  scontext=system_u:system_r:syslogd_t \
  tcontext=system_u:object_r:bin_t tclass=file
Jan 10 16:20:35 example kernel: audit(1009284205.423:0): \
  avc:  denied  { read } for  pid=1411 exe=/bin/bash \
  path=/bin/rm dev=dm-0 ino=1245243 \
  scontext=system_u:system_r:syslogd_t \
  tcontext=system_u:object_r:bin_t tclass=file
```

Running all of the audit messages through `audit2allow` generates a set of rules:

```
cd  /etc/selinux/targeted/src/policy/domains/misc/
audit2allow -i /var/log/messages -o ./local.te
cat local.te
allow syslogd_t bin_t:dir search;
allow syslogd_t bin_t:file { execute execute_no_trans getattr \
  read };
allow syslogd_t bin_t:lnk_file read;
allow syslogd_t etc_runtime_t:file { getattr read };
allow syslogd_t proc_kmsg_t:file write;
allow syslogd_t proc_t:file { getattr read };
allow syslogd_t sbin_t:dir search;
allow syslogd_t shell_exec_t:file { execute execute_no_trans \
  getattr read };
allow syslogd_t self:capability { chown fowner fsetid sys_admin };
allow syslogd_t usr_t:dir { add_name remove_name write };
allow syslogd_t usr_t:file { append create getattr read setattr \
  unlink write };
```

Looking at the rules, you can see that there are two for execution permissions, and some other rules that are associated by having the same object, `bin_t`. There is also a permission to search directories of the type `sbin_t`, but no execution permissions.

From your reading of the available macros in `$SELINUX_SRC/macros/`, you know that `can_exec()` provides a common set of permissions for domains wishing to execute certain file types. This includes permissions that are likely to arise once the first set of basic rules are used. For example, after `audit2allow` generates a rule giving read permission to a process, the process often wants `getattr` permission. The can_exec() macro includes the permission rx_file_perms, which grants a common set of read and execute permissions to a file. Now you can make this substitution:

```
# these related rules ...
allow syslogd_t bin_t:dir search;
allow syslogd_t bin_t:file { execute execute_no_trans getattr \
  read };
allow syslogd_t shell_exec_t:file { execute execute_no_trans \
  getattr read };

# combine into this one rule
can_exec(syslog_t, { bin_t shell_exec_t } )
```

You also see in the rules that two of the rules are from syslog-ng attempting to use a directory in /usr/ of the type usr_t:

```
# These rules can be eliminated by properly labeling
# the files in the target location.
allow syslogd_t usr_t:dir { add_name remove_name write };
allow syslogd_t usr_t:file { append create getattr read setattr \
  unlink write };
```

Your configuration uses a directory in /usr/ to write log files to. Because this log data is not user data, it should have an appropriate label, var_log_t.

```
# If syslog-ng is configured to put logs in /usr/local/logs/,
# relabel that directory, and new files in the directory
# inherit the proper type.
chcon -R -t var_log_t /usr/local/logs/
```

Now you can trim the rules in $SELINUX_SRC/domains/misc/local.te to read:

```
can_exec(syslog_t, { bin_t shell_exec_t } )
allow syslogd_t etc_runtime_t:file { getattr read };
allow syslogd_t proc_kmsg_t:file write;
allow syslogd_t proc_t:file { getattr read };
allow syslogd_t bin_t:lnk_file read;
allow syslogd_t sbin_t:dir search;
allow syslogd_t self:capability { chown fowner fsetid sys_admin };
```

You also need to make an appropriate file contexts file so that the labeling is maintained during relabeling operations. Put the following context declaration in /etc/selinux/targeted/src/policy/file_contexts/misc/local.fc:

```
/usr/syslog(/.*)?   system_u:object_r:var_log_t
```

## 8.3. Writing New Policy for a Daemon

These section provides an overall methodology to follow for writing a new policy from scratch. Although this is more complex than adding a few rules to local.te, the concepts are the same. You bring the application under TE rules and work through the AVC denials, adding rules each time until all permissions are resolved.

### New Policy Writing Procedure

1. Work with a proper daemon under Red Hat Enterprise Linux. This means it has an initscript in `/etc/init.d/` and can be managed using `chkconfig`. For example, this procedure assumes you are going to use the `service` command to control starting and stopping the daemon.

   For this procedure, you are writing policy for the fictional `foo` package and it's associated `foo` daemon. Use a real daemon name in this place when developing your own policy.

2. Create a file at `$SELINUX_SRC/domains/program/foo.te`.
3. Put the daemon domain macro call in the file:
   ```
   daemon_domain(foo)
   ```
4. Create the file contexts file, `$SELINUX_SRC/file_contexts/program/foo.fc`.
5. Put the first list of file contexts in `file.fc`. You may need to add to this later, depending on the needs of the `foo` daemon.
   ```
   /usr/bin/foo      --     system_u:object_r:foo_exec_t
   /var/run/foo.pid  --     system_u:object_r:foo_var_run_t
   /etc/foo.conf     --     system_u:object_r:foo_conf_t
   ```
6. Load the new policy with `make load`.
7. Label the `foo` files:
   ```
   restorecon /usr/bin/foo /var/run/foo.pid /etc/foo.conf
   ```
8. Start the daemon, `service foo start`.
9. Examine your audit log for denial messages:
   ```
   grep "avc:  denied" /var/log/messages > /tmp/avc_denials
   cat /tmp/avc_denials
   ```

   Familiarize yourself with the errors the daemon is generating. You are writing policy with the help of `audit2allow`, but you need to understand the nature of the denials. You can also use **seaudit** for viewing the log messages, as explained in Section 6.2 *Using **seaudit** for Audit Log Analysis*.

10. Use `audit2allow` to start the first round of policy rules.
    ```
    audit2allow -l -i /var/log/messages -o \
      /etc/selinux/targeted/src/policy/domains/program/foo.te
    ```

    When looking at the generated rules, if you see a rule that gives the `foo_t` domain `read` access to a file or directory, change the permission to read `{ read gettatr }`. The domain is likely to need that permission if it already wants to read a file.

11. Look to see if the `foo_t` domain tries to create a network socket, that is, `udp_socket` or `tcp_socket` as the object class in the AVC denial:
    ```
    avc:  denied  { create } for  pid=7279 exe=/usr/bin/foo \
      scontext=root:system_r:foo_t tcontext=root:system_r:foo_t\
      tclass=udp_socket
    ```

    If this is the case, then add the `can_network()` macro to `foo.te`:
    ```
    can_network(foo_t)
    ```

12. Continue to iterate through the basic steps to generate all the rules you need. Each set of rules added to the policy may reveal additional permission needs from the `foo_t` domain.

    a. Start the daemon.
    b. Read the AVC messages.
    c. Write policy from the AVC messages, using `audit2allow` and your own knowledge, looking for chances to use macros.
    d. Load new policy.
    e. Go back to beginning, starting the daemon ...

13. If the domain tries to access `port_t`, which relates to `tclass=tcp_socket` or `tclass=udp_socket` in the AVC log message, you need to determine what port number `foo` needs to use. To diagnose, put these rules in `foo.te`:

    ```
    allow foo_t port_t:tcp_socket name_bind;
    auditallow foo_t port_t:tcp_socket name_bind;
    ```

    The `auditallow` rule helps you determine the nature of the port connection attempt.

14. Iterate through the remaining AVC denials. When they are resolved with new policy, you can configure the unique port requirements for the `foo_t` domain.

15. With the daemon started, determine which port `foo` is using. Look at the AVC allowed message and see what port the daemon is connected to:

    ```
    lsof | grep foo.*TCP
    foo   2283  root   3u  IPv6   3192    TCP *:4242 (LISTEN)
    ```

    The `foo` daemon is listening on port 4242.

16. Remove the generic `port_t` rule, replacing it with a specific rule for a new port type based on the `foo_t` domain.

    ```
    type foo_port_t, port_type;
    allow foo_t foo_port_t:tcp_socket name_bind;
    ```

    Add this line to `$SELINUX_SRC/file_contexts`. This reserves the port 4242 for the domain `foo_t`:

    ```
    ifdef(‘foo.te’, ‘portcon tcp 4242 system_u:object_r:foo_port_t’)
    ```

## 8.4. Deploying Customized Binary Policy

Building custom binary policy requires the policy source, but there are security risks to having the full policy source on a production server. Should an attacker gain root control, they could rebuild the policy to weaken or neutralize SELinux. For this reason, you want to use only binary policy on production servers.

If you are using binary policy provided by Red Hat, this is not an issue. You can manage the policy packages as you do any other software packages, such as through Red Hat Network. If you do customize your policy, you may want to manage it using RPM packages, either manually or as part of custom channels in Red Hat Network. Even if you manage the policy files by hand, as this procedure demonstrates, the underlying principles are the same.

Unlike software source code, SELinux policy is independent of the architecture of the system it is built on. You can have multiple development trees with vastly different policy needs expressed, all residing and built in a single development environment. In fact, it is recommended to use software version control, such as CVS, if you are going to be doing any measurable level of customization.

This short procedure demonstrates how to build and deploy policy from a development environment into a production environment.

### Building and Deploying Policy

1. In your development environment, make a copy of the source tree for the policy you are working from. If you want to add a single daemon to confinement under the targeted policy, use `/etc/selinux/targeted/src/policy/`. If you want to work from a fully confined SELinux environment, obtain and use the strict source, which is usually at `/etc/selinux/strict/`.

    ```
    # Copy the entire tree to a custom/ directory:

    cp –r /etc/selinux/targeted/ /etc/selinux/custom/
    ```

2. Build and test your policy. You can test locally on your development machine, or follow the outline of this procedure to deploy custom binary policy files to your test environment. Use **apol** to analyze your policy, as described in Section 6.3 *Using **apol** for Policy Analysis*.

3. When you are ready to deploy, use `tar` to pack your policy files. Notice that the source directory is not included.

   ```
   tar -czvf tgt.tgz targeted/policy/ targeted/contexts/ \
     targeted/booleans
   ```

   Alternately, you can use `star` so that you can preserve the xattrs for the policy files. This is explained in Section 5.1.4 *Make Backups or Archives That Retain Security Contexts*. However, since you are going to initiate a relabel on boot anyway, you can use `tar` instead. The policy files unpack and gain the default file label such as `system_u:object_r:default_t` and are relabeled upon boot.

4. If this is the first time the custom policy has been deployed on this system, you need to configure SELinux to use the policy on the next boot.

   **Note**

   It is extremely difficult to change policy without rebooting the system. The file system needs to be relabeled and every process starting with `init` needs to be restarted under the new policy. This is the reason rebooting is required for switching policy.

   In `/etc/selinux/config`, change the value for `SELINUXTYPE` to the name of the new policy. The name is the same as the directory name in `/etc/selinux`. For example, the custom policy at `/etc/selinux/custom/` has the value of `SELINUXTYPE=custom`.

   You can do this using **system-config-securitylevel**. Under the **SELinux** tab, change the **Policy Type:** to **custom**. This area of **system-config-securitylevel** is automatically populated from the names of actual policy directories under `/etc/selinux/`.

5. Initiate a reboot and relabel.

   ```
   touch /.autorelabel
   reboot
   ```

6. If you have troubles getting the custom policy to work on the test or production environment, work through the denials like you did when writing the policy in the first place.

   a. Make sure the file system is labeled correctly. If you cannot `touch /.autorelabel`, either use `setenforce 0` or boot into permissive mode.

      You may need to boot into single-user mode and attempt a manual relabel of the file system. Although this is not normally recommended, it can be a working method to get enough labeling correct to have the `/.autorelabel` work correctly. You can read more about this at Section 5.2.2 *Relabel a File System*.

   b. Work through the denial errors one at a time. You may need to temporarily install the policy source to relabel or rebuild the policy.

# Chapter 9.

# References

The following references are pointers to additional information that is relevant to SELinux and Red Hat Enterprise Linux but beyond the scope of this guide.

## Tutorials and Help

Understanding and Customizing the Apache HTTP SELinux Policy

http://fedora.redhat.com/docs/selinux-apache-fc3/

Tutorials and talks from Russell Coker

http://www.coker.com.au/selinux/talks/ibmtu-2004/

Generic Writing SE Linux policy HOWTO

https://sourceforge.net/docman/display_doc.php?docid=21959&group_id=21266

Red Hat Knowledgebase

http://kbase.redhat.com/

## General Information

NSA SELinux main website

http://www.nsa.gov/selinux/

NSA SELinux FAQ

http://www.nsa.gov/selinux/info/faq.cfm

Fedora SELinux FAQ

http://fedora.redhat.com/docs/selinux-faq-fc3/

SELinux NSA's Open Source Security Enhanced Linux

http://www.oreilly.com/catalog/selinux/

## Technology

An Overview of Object Classes and Permissions

http://www.tresys.com/selinux/obj_perms_help.html

Integrating Flexible Support for Security Policies into the Linux Operating System (a history of Flask implementation in Linux)

http://www.nsa.gov/selinux/papers/slinux-abs.cfm

Implementing SELinux as a Linux Security Module

http://www.nsa.gov/selinux/papers/module-abs.cfm

A Security Policy Configuration for the Security-Enhanced Linux

http://www.nsa.gov/selinux/papers/policy-abs.cfm

## Community

SELinux community page

http://selinux.sourceforge.net

IRC

irc.freenode.net, #rhel-selinux

## History

Quick history of Flask

http://www.cs.utah.edu/flux/fluke/html/flask.html

Full background on Fluke

http://www.cs.utah.edu/flux/fluke/html/index.html

# III. Appendix

## Table of Contents

# Appendix A.

# Brief Background and History of SELinux

SELinux was originally a development project from the National Security Agency (*NSA*)[1] and others. It is an implementation of the *Flask* operating system security architecture[2]. The Flask architecture implements MAC, which focuses on providing an administratively-defined security policy that can control all subjects and objects, basing decisions on all security-relevant information. In addition, Flask focuses on the concept of *least privilege*, which gives a process exactly the rights it needs to perform it's given task.

The Flask model allows you to express a security policy in a naturally flowing manner, so that parts of the security rules are like parts in a sentence. In Flask, changes are supported so you can tune your policy. Added to this architecture in the security server are TE and RBAC security models, providing fine-grained controls that can be transparent to users and applications.

As a next step in the evolution of SELinux, the NSA integrated SELinux into the Linux kernel using the *Linux Security Modules* (*LSM*) framework. SELinux motivated the creation of LSM, at the suggestion of Linus Torvalds, who wanted a modular approach to security instead of accepting just SELinux into the kernel.

Originally, the SELinux implementation used *persistent security IDs* (PSIDs) stored in an unused field of the ext2 inode. These numerical representations (i.e., non-human-readable) were mapped by SELinux to a security context label. Unfortunately, this required modifying each file system type to support PSIDs, so was not a scalable solution or one that would be supported upstream in the Linux kernel.

The next evolution of SELinux was as a loadable kernel module for the 2.4.<*x*> series of Linux kernels. This module stored PSIDs in a normal file, and SELinux was able to support more file systems. This solution was not optimal for performance, and was inconsistent across platforms. Finally, the SELinux code was integrated upstream to the 2.6.*x* kernel, which has full support for LSM and has *extended attributes* (*xattrs*) in the ext3 file system. SELinux was moved to using xattrs to store security context information. The xattr namespace provides useful separation for multiple security modules existing on the same system.

Much of the work to get the kernel ready for upstream, as well as subsequent SELinux development, has been a joint effort between the NSA, Red Hat, and the community of SELinux developers.

For more information about the history of SELinux, the definitive website is http://www.nsa.gov/selinux/.

[1] The NSA is the cryptologic agency of the United States of America's Federal government, charged with information assurance and signals intelligence. You can read more about the NSA at their website, http://www.nsa.gov/about/.

[2] Flask grew out of a project that integrated the *Distributed Trusted Operating System* (*DTOS*) into the Fluke research operating system. Flask was the name of the architecture and the implementation in the Fluke operating system.

# Index

# Colophon

The manuals are written in DocBook SGML v4.1 format. The HTML and PDF formats are produced using custom DSSSL stylesheets and custom jade wrapper scripts. The DocBook SGML files are written in **Emacs** with the help of PSGML mode.

Garrett LeSage created the admonition graphics (note, tip, important, caution, and warning). They may be freely redistributed with the Red Hat documentation.

The Red Hat Product Documentation Team consists of the following people:

Sandra A. Moore — Primary Writer/Maintainer of the *Red Hat Enterprise Linux Installation Guide for x86, Itanium™, AMD64, and Intel® Extended Memory 64 Technology (Intel® EM64T)*; Primary Writer/Maintainer of the *Red Hat Enterprise Linux Installation Guide for the IBM® POWER Architecture*; Primary Writer/Maintainer of the *Red Hat Enterprise Linux Installation Guide for the IBM® S/390® and IBM® eServer™ zSeries® Architectures*

John Ha — Primary Writer/Maintainer of the *Red Hat Cluster Suite Configuring and Managing a Cluster*; Co-writer/Co-maintainer of the *Red Hat Enterprise Linux Security Guide*; Maintainer of custom DocBook stylesheets and scripts

Edward C. Bailey — Primary Writer/Maintainer of the *Red Hat Enterprise Linux Introduction to System Administration*; Primary Writer/Maintainer of the *Release Notes*; Contributing Writer to the *Red Hat Enterprise Linux Installation Guide for x86, Itanium™, AMD64, and Intel® Extended Memory 64 Technology (Intel® EM64T)*

Karsten Wade — Primary Writer/Maintainer of the *Red Hat SELinux Guide*; Contributing Writer to the *Red Hat Enterprise Linux System Administration Guide*

Andrius T. Benokraitis — Primary Writer/Maintainer of the *Red Hat Enterprise Linux Reference Guide*; Co-writer/Co-maintainer of the *Red Hat Enterprise Linux Security Guide*; Contributing Writer to the *Red Hat Enterprise Linux System Administration Guide*

Paul Kennedy — Primary Writer/Maintainer of the *Red Hat GFS Administrator's Guide*; Contributing Writer to the *Red Hat Cluster Suite Configuring and Managing a Cluster*

Mark Johnson — Primary Writer/Maintainer of the *Red Hat Desktop Deployment Guide*

Melissa Goldin — Primary Writer/Maintainer of the *Red Hat Enterprise Linux Step By Step Guide*

Lucy Ringland — Red Hat Enterprise Linux Documentation Editor.

The Red Hat Localization Team consists of the following people:

Amanpreet Singh Alam — Punjabi translations

Jean-Paul Aubry — French translations

David Barzilay — Brazilian Portuguese translations

Runa Bhattacharjee — Bengali translations

Chester Cheng — Traditional Chinese translations

Verena Fuehrer — German translations

Kiyoto Hashida — Japanese translations

N. Jayaradha — Tamil translations

Michelle Jiyeen Kim — Korean translations

Yelitza Louze — Spanish translations

Noriko Mizumoto — Japanese translations

Ankitkumar Rameshchandra Patel — Gujarati translations

Rajesh Ranjan — Hindi translations

Nadine Richter — German translations
Audrey Simons — French translations
Francesco Valente — Italian translations
Sarah Wang — Simplified Chinese translations
Ben Hung-Pin Wu — Traditional Chinese translations